KUNGL TEKNISKA HÖGSKOLAN

Royal Institute of Technology

MASTER OF SCIENCE THESIS

Video Coding in H.26L

by

Kristofer Dovstam

April 2000

Work done at Ericsson Radio Systems AB, Kista, Sweden,
Ericsson Research, Department of Audio & Visual Technology.

Abstract

For the past few years, the capacity of global networks and communication channels has increased considerably. This allows for real-time applications like video conferencing and video-on-demand using compressed video. State-of-the-art video coding solutions such as H.263, MPEG-2 and MPEG-4 all have one goal in common: to achieve highest possible image quality for lowest possible bit-rate.

During 1999, the development of a new video coding standard, H.26L, started. H.26L is supposed to replace its predecessor H.263, and one of the goals is to achieve 50% greater bit-rate savings compared to H.263. The first proposal for the H.26L standard, presented in August 1999, achieves higher compression efficiency compared to H.263. However, the goal of 50% is not yet met.

This thesis proposes a method to use adaptive arithmetic coding (AAC) for entropy coding in an H.26L video codec in order to further improve the compression efficiency. It is a general solution that can be applied to any video codec. However, implementations have been made for the H.26L video codec only. AAC is based on an entirely different strategy than the variable length entropy coding technique employed in H.26L and many other video codecs.

Three experimental models for adaptation to local statistics have been designed and investigated. Results show that for high bit-rate environments significant bit-rate savings can be made using AAC, while less can be won for lower bit-rates.

Contents

1. Introduction and Background

This section includes an overview of video compression, a brief introduction to current video coding standards, and a discussion about video coding research topics. The experienced reader, who is familiar with concepts such as motion estimation, adaptive arithmetic coding and error resilience could skip to the purpose and goal in Section 1.4.

1.1 Overview of Video Compression

The key idea of video compression is to exploit spatial and temporal redundancy. Spatial redundancy exists within an image due to similarities between different areas of the image. For instance, adjacent pixels (picture elements) in an image are very likely to have similar characteristics. Temporal redundancy can be found within a sequence of frames (video) and is due to similarities between successive frames. (Throughout this document the words image, frame and picture will be used interchangeably.) Procedures for exploiting redundancy have different approaches depending on the redundancy being spatial or temporal.

There are two different kinds of compression: lossless compression and lossy compression, also referred to as reversible and irreversible compression. If data is compressed using a lossless compression technique, the information content of the original data and the compressed data is exactly the same. On the other hand, if a lossy compression technique is employed, the compressed data will contain less information than the original data, i.e., information is lost forever.

It is important to remember that when dealing with discrete representation of information, such as in a computer, data will always be quantized. This means that data-values, for example pixels, will always be rounded off or truncated to fit a certain number of bits. The fewer available bits, the coarser the quantization. If, for example, a natural image is quantized, the resulting quantized image always contains less information than the natural image.

An important property of a compressed video signal is its bit-rate. The bit-rate is the average number of bits needed to represent each second of a video sequence, and the unit is *bits per second*. In other words, the bit-rate of a video sequence is its size in bits divided by its length in seconds.

In video compression in general, the goal is to obtain a lower bit-rate for the compressed video signal than for the same uncompressed signal. It is, however, important to consider factors other than bit-rate as well. If, for instance, video is used in mobile communications, the compressed video signal needs to be insensitive to bit-errors. That is, the video signal needs to be error resilient. It is also preferable to keep down the complexity of the encoder and decoder and to add as little delay to the whole system as possible.

1.1.1 Exploiting Spatial Redundancy – INTRA-coded Compression

The most popular approach for removing spatial redundancy in an image is by means of a decorrelating transform, which decreases the correlation between pixels, and hence represents the information in the image in a more compact way. The transform is usually applied on square blocks of the image (4x4, 8x8, 16x16 pixels, etc). After transformation the block consists of transform coefficients containing the same information as the pixels. The transform coefficients correspond to different frequencies.

The transformation does not in itself result in any compression. Instead, compression is achieved when the transform coefficients are quantized. The coarser the quantization, the higher the compression. This also results in loss of information. If pixels are quantized, all the frequencies in the image are affected equally much. However, when transform coefficients are quantized, the higher frequencies, for which the human eye is less sensitive, are often more coarsely quantized than the lower frequencies. This results in a smaller degradation of the apparent image quality than if the transform coding technique is not employed and the quantization is performed directly on the pixels. When decoding, the transform coding process is reversed and the pixel values are restored using an inverse transform. In video coding, when the encoding of an image in a sequence is independent of other frames in the sequence, the process is called INTRA-coded compression. The image is referred to as an INTRA-picture or I-picture.

Examples of transforms with good decorrelating performance are the often adopted discrete cosine transform (DCT) and wavelet-transforms. As of today the most commonly used transform in video coding is the DCT, while for example the JPEG-2000 [7] image compression standard uses wavelet-transforms.

1.1.2 Exploiting Temporal Redundancy – INTER-coded Compression

Typically, adjacent images in a sequence of frames, for example video with a frame-rate of 30 frames/s, are very similar. If we encode every frame as an INTRA-picture, we pay no attention to the fact that there is much correlation between consecutive frames (an example of an algorithm taking this approach is the Motion-JPEG algorithm (cf. [1])). The obvious solution to this problem is to represent only the difference for each frame with respect to the previous frame. This can easily be done by simply subtracting the pixel values in the frame currently being encoded with the pixel values from the previous frame. In this way, the pixel value variance between frames decreases and a stronger compression can be obtained. In the next step, the same transform compression technique as for INTRA-pictures is used on the difference data. Further compression can be achieved, for example by introducing a threshold. If the difference in pixel values between corresponding consecutive pixels in frames is below this threshold, we will ignore the difference and simply use the previous pixel values. In video coding, when an image in a sequence is encoded this way, the process is called INTER-coded compression.

1.1.3 Motion Estimation

INTER-coded compression can be improved by motion estimation. One of the simpler and less complex forms of motion estimation is forward prediction using block matching. This approach is described below.

In current major video coding standards, images are partitioned into blocks called macroblocks. The size of a macroblock is typically 16x16 pixels. When a video sequence is encoded we predict the pixel values for each macroblock in each frame. Here, the macroblock to be predicted is called the target macroblock. The prediction is done by looking at a past frame called the reference picture. When encoding a frame, if no movement has taken place with respect to the reference picture, all the target macroblocks will be best predicted by the corresponding macroblock in the reference picture. If, however, there is movement, one or more of the target macroblocks may be better predicted by a displaced but equally sized block. This displacement can be found by matching the target macroblock with an appropriately positioned set of blocks of the same size (a search area) in the reference picture (see Figure 1.1). The "best" matching block will be used to predict the target macroblock.

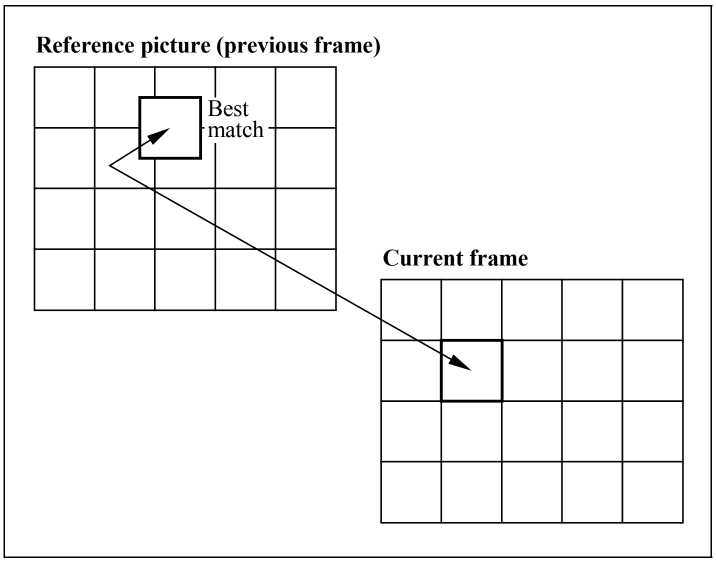

Figure 1.1 Forward prediction using block matching

A commonly adopted metric when searching for the best match is the well-known mean square error (MSE) (see Section 2.1.1 for a definition of MSE). The smaller the error the better the match. In practical video coding, however, a simplified version of the MSE called sum of absolute differences (SAD) [1] is often employed. This decreases the amount of computations considerably since no multiplications are needed to compute the SAD.

When the best matching block is found, a motion vector is designed to represent the displacement of the block, and the prediction error is calculated. The prediction error is the difference in pixel values between the target macroblock and the block used for prediction. Now, in order to decode a macroblock, all we need is its motion vector and the prediction error.

A frame coded by using forward prediction is referred to as a P-picture. There exist far more advanced and sophisticated methods for motion estimation of which some will be discussed in Section 1.3.3.

1.1.4 Entropy Coding

When all frames in the sequence have been INTRA- or INTER-coded, the video signal consists of transform coefficients and motion vectors (a few bits are also needed for quantization information, type of macroblock encoding, etc). Due to statistical properties, however, there is still redundancy in the video signal. This redundancy can be exploited by adopting an entropy coding technique. Contrary to compression techniques discussed in earlier sections, entropy coding is a lossless stage of the video coding procedure.

The video signal can be viewed as a source of information, which has a *true* entropy or information content (see Section 2.5.2 for a definition of entropy). By modeling the information source, we assign probabilities to different outcomes, or symbols, of the source. In video coding the symbols are typically the transform coefficients and the motion vectors. After modeling comes coding, which, based on the probabilities assigned by the model, translates the symbols to a sequence of bits or a bit-stream.

Run-Length Coding

An easy way to represent symbols efficiently is to use run-length coding. If the symbols in a serie of successive symbols in a signal are the same, we replace the symbols with one symbol plus the length of the run of that symbol. Provided many successive symbols are the same, this will result in a considerably decreased amount of bits necessary to represent the data. The technique can also be easily combined with other entropy coding techniques.

Variable Length Codes

By far the most commonly adopted entropy coding technique for video coding purposes is variable length codes (VLCs). Examples of VLCs are the well-known Huffman codes (cf. [3]). In the VLC approach, compression is achieved by assigning shorter codewords, i.e. short sequences of bits, to symbols with higher probabilities and longer codewords to symbols with lower probabilities. The codewords and associated symbols are organized in what is called a VLC table or a look-up table. There is, however, a limitation to how accurate the probabilities can be represented, which is due to the fact that each VLC is constrained to consist of an integral number of bits.

Arithmetic Coding

Another approach to entropy coding, quite different from VLCs and Huffman coding, is arithmetic coding (cf. [3] [5] [8]). In this approach, after modeling the information source, it is possible to represent the probabilities exactly. A sequence of symbols, or a message, is, after being encoded with the arithmetic coding technique, represented by an interval of real numbers between 0 and 1. The coding procedure is easiest described using an example (similar to the one found in [3]).

Assume that we have an alphabet of three symbols with corresponding probabilities as in Table 1.1a. At first, each symbol is assigned a unique interval in between 0 and 1 according to its probability (see Table 1.1b). Let us say that we want to encode the message *acab*. The procedure is illustrated in Figure 1.2a and 1.2b. The message, i.e. the stream of symbols (*acab*), is put into the encoder. When the encoder "sees" the first symbol *a* in the message, *a*'s interval, i.e. [0, 0.5), is chosen to represent the entire message, which so far is only *a*. However, the message is longer than just *a*, and when the encoder sees the next symbol *c*, all

Table 1.1a
Symbols and probabilities

Symbol	Probability
a	0.5
b	0.2
c	0.3

Table 1.1b
Symbols and intervals

Symbol	Interval
a	[0, 0.5)
b	[0.5, 0.7)
c	[0.7, 1.0)

the probabilities, and hence all the intervals, are normalized, or scaled, so that they all lie in the interval of *a* (i.e. in between 0 and 0.5 according to Figure 1.2a and 1.2b). By doing so, it is possible to represent *c* **and** *a* with *c*'s newly assigned interval, [0.35, 0.5). As seen in

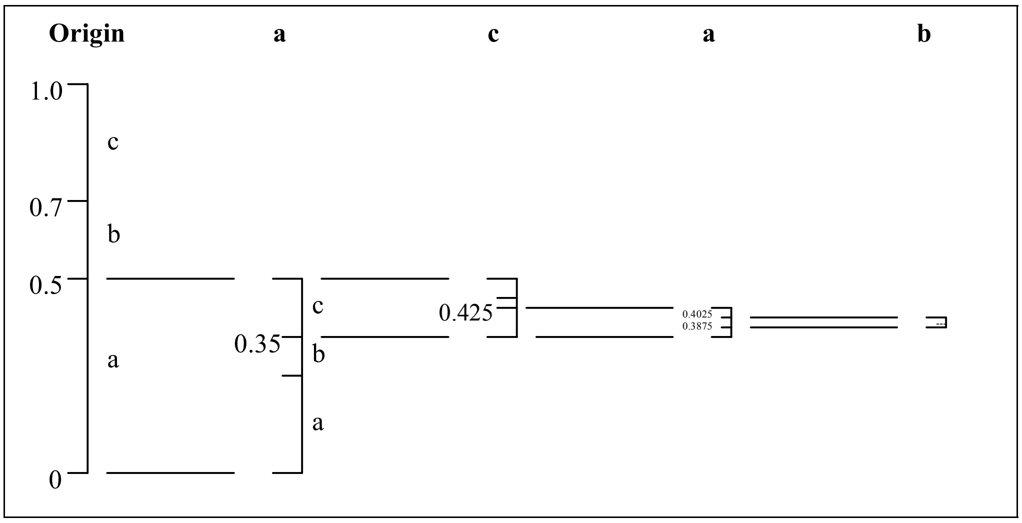

Figure 1.2a The process of dividing intervals into smaller subintervals.

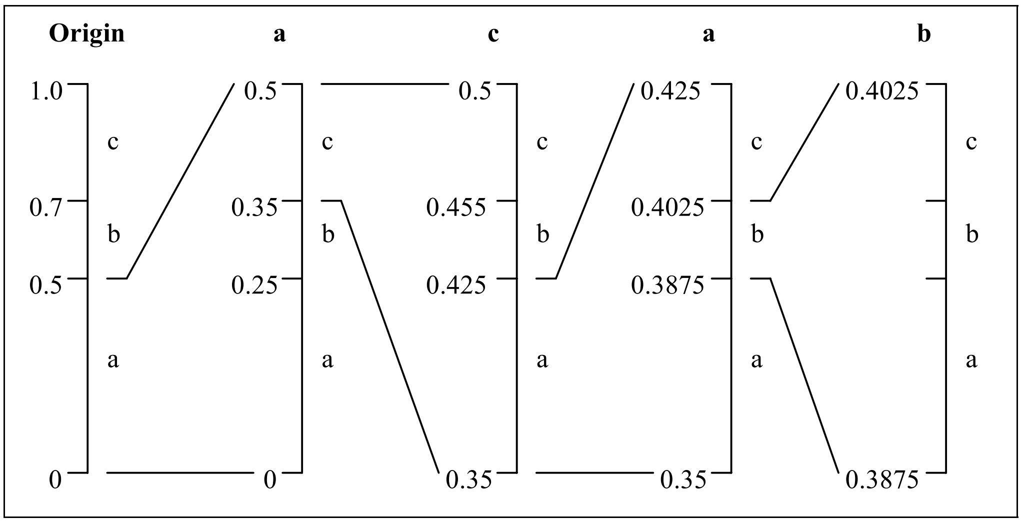

Figure 1.2b Same process as in Figure 1.2a, here each interval is expanded to full height.

Figure 1.2a, the interval [0.35, 0.5) is unique for the message *ac*. Now, the rest of the symbols will be encoded in the same manner within the interval [0.35, 0.5) and so on. The same process in each step assures that the message "seen so far" always can be uniquely represented. More probable symbols will reduce the size of the interval less, adding fewer bits to the message. The longer the message, the finer the last interval becomes and the more bits are needed to represent it. When we have reached the last symbol *b*, the message *acab* is represented by the interval [0.3875, 0.4025). However, it is not necessary for the decoder to know both ends of the interval, a number lying in the interval will suffice, such as 0.4 or 0.3879 or even 0.401345978. For decoding it is also necessary to have knowledge of the source model, i.e. the symbols and probabilities in Table 1.1a.

The decoding process is essentially a reversed encoding process. However, during decoding we are faced with a problem; how do we know when to stop decoding? Indeed, any real number in between 0 and 1 could represent an infinitely long message, since it is always possible to divide an interval into new subintervals. This problem is solved by adding an *end of message* symbol to the alphabet that will be used **only** to terminate messages. When the decoder encounters the end of message symbol, it knows that the entire message has been decoded and stops decoding. In the example above, we could choose *b* to be our end of message symbol.

Another approach to solving the problem of knowing when to stop decoding is to make use of contexts. For a particular information source it may be clear within different contexts how many symbols are to be encoded and decoded. If this is the case, the decoder knows from the context when to stop decoding, and does not need an end of message symbol. Let us say, for instance, that the encoder and decoder agree to send messages with only three symbols. Then, when the decoder in the example above has determined the first three symbols *aca*, it knows that it is time to stop decoding and there is no need for the terminating *b*.

When arithmetic coding is implemented on a computer, the number representing the message is in turn binary represented and sent to the decoder as a bit-stream. There are two concerns, however, regarding the implementation of arithmetic coding on a computer. First of all, not very much of a message need to be encoded before the interval [0,1) has been divided into a sub-interval too small to be represented on a state of the art computer. Therefore, we need to decide the precision to which the arithmetic coding is performed. Second, the scheme described above encodes the entire message before any bits are sent to the decoder. This is not acceptable in a real time environment, which calls for incremental transmission and reception.

Incremental transmission is easily implemented by realizing that the first decimal in the two numbers forming the interval that represents the message will not change after a series of symbols have been encoded. For instance, if the interval representing the message so far is [0.358, 0.364), we know that the entire message will be represented by a number starting with 0.3... and so, we can send information about the 0.3 and concentrate on the interval [0.058, 0.064). The latter interval forces us to look at one or more of the following symbols in the message to see if the next decimal is a 5 or a 6. In the computer, the procedure is performed using binary representation, i.e. with base 2 instead of base 10.

The problem concerning precision in the encoder is solved by periodically normalizing, or scaling the interval representing the message. In the procedure above, after sending information about the 0.3, the interval is scaled, or multiplied by 10 to form the interval [0.58, 0.64). The interval is multiplied by 10 because that is the base of the representation, and hence, in a computer using binary representation, the interval is instead multiplied by 2.

When decoding incrementally, an interval is kept in the decoder corresponding to the precision of the number that has been received from the bit-stream so far. For example, when information about the 0.3 has been received, we know that the message so far is represented by a number in the interval [0.3, 0.4). In order to decode a symbol uniquely we need to receive a certain amount of decimals, or equivalently, bits in the binary representation case. The number of bits needed is dependent on the size of the interval (or probability) assigned to the symbol that the bits belong to. A symbol can not be uniquely decoded until the interval kept in the decoder lies entirely inside the symbol's interval. As mentioned earlier, probable symbols with a large interval need fewer bits to be uniquely decoded while less probable symbols with small intervals need more bits to be uniquely decoded. Instead of allowing infinitely small intervals, demanding infinitely many bits for decoding, we decide that B bits is enough to uniquely decode a symbol, and by doing so put a limitation on how small a symbol interval can be. B is the precision of the arithmetic codec, and the precision of the numbers forming a symbol's interval needs to be somewhat lower than B so that B can represent a number lying **inside** the smallest interval. When the decoding starts, we immediately fill up a buffer, or a window of B bits from the bit-stream. Bits are then processed by moving them out of the high-order end of the window until a symbol can be uniquely decoded. As bits are moved out from the window, new bits are read from the bit-stream and shifted into the low-order end of the window. When a symbol has been identified, the interval kept in the decoder is normalized, or scaled in the same manner as in the encoder.

Not mentioned so far is that when implementing arithmetic coding, it would in many aspects be very inefficient to represent intervals using floating-point numbers. Instead, all the intervals mentioned above are multiplied by the precision B of the codec, allowing them to be represented using integers.

It is a fact that arithmetic coding produces a code with an entropy closer to that of the message than does variable length codes or Huffman codes. Still, as stated earlier, VLCs are more often adopted in video coding. One probable reason for this is that it is believed that arithmetic coding produces a less error resilient bit-stream, i.e., in case of bit-errors in the bit-stream, arithmetic coding would do worse then VLCs.

A drawback of arithmetic coding is the large amount of computations needed for both encoding and decoding. Various solutions to this problem have been proposed. In, for example, [10] a method to approximate binary arithmetic coding is presented. In this approach, computationally expensive multiplication and division operations are replaced by a small number of shifts, additions and subtractions.

1.1.5 A Typical Video Codec

With the basic video coding tools introduced thus far, it is possible to construct a simple video codec (video coder and decoder). Figure 1.3 illustrates a typical video codec that uses the DCT for transform coding. As seen in Figure 1.3a, the coder needs to inherently contain a decoder in order to compensate for motion correctly. It is important that the inverse transform in the coder matches the inverse transform in the decoder (no mismatch). Otherwise, the prediction error (difference data), which is based on inverse transformed information obtained in the coder, will not be the correct prediction error for the decoder using its own inverse transform.

For further readings about general video coding, refer to [1] [23] [26] [29].

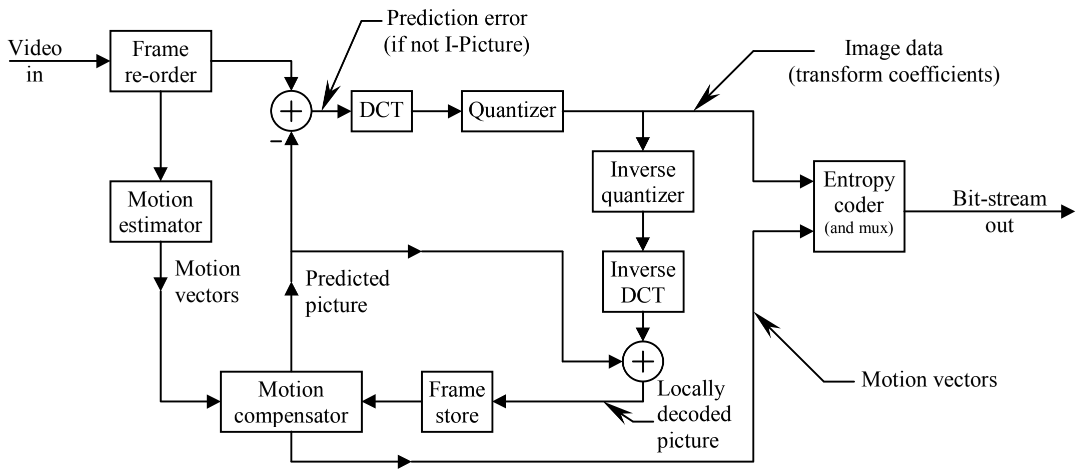

Figure 1.3a Typical video coder.

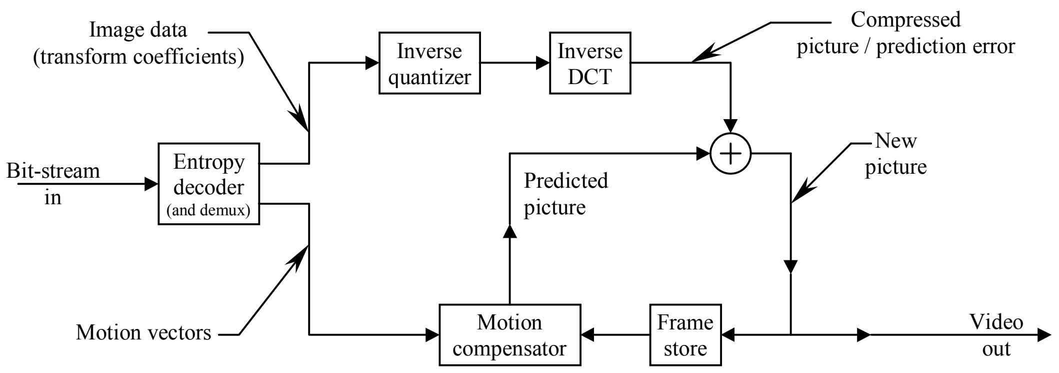

Figure 1.3b Typical video decoder.

1.2 A Brief Overview of Existing Standards

Since 1984, when the development of H.261 (Section 1.2.1) started, a number of new standards presenting new ideas, algorithms and solutions have been developed. There are two main video coding standardization organizations, International Telecommunication Union – Telecommunication Standardization Sector (ITU-T) and International Organization for Standardization / International Electrotechnical Commission (ISO/IEC), which both have their own series of video compression standards. ITU-T is developing the H.26x recommendations, which are optimized for low bit-rates. ISO/IEC is responsible for the Moving Picture Experts Group (MPEG) standards. This is a larger standard that represents not only coded video but also associated coded audio, synchronization and multiplexing, etc. Here, however, only the video coding parts of the MPEG standards, as well as the ITU-T video coding recommendations will be considered.

1.2.1 H.261

The first recommendation from ITU-T in the H.26x series was H.261 (cf. [23]). The approach to video coding is similar to general video coding described in Section 1.1. It employs the DCT for transform coding and VLCs for entropy coding. H.261 is designed for real-time applications and optimized for low bit-rates (down to 64 kbits/s). A simple motion estimation technique reduces complexity and encoding time. H.261 is a compromise between coding performance, real-time requirements, implementation complexity, and system robustness.

1.2.2 MPEG-1

Not very long after the introduction of H.261, MPEG-1 (cf. [23]) emerged, which is a derivative of H.261. The primary objective of MPEG-1 is to achieve highest possible image quality at a given bit-rate. During the development, however, the standard was optimized for a bit-rate of 1.1 Mbits/s.

A new feature in MPEG-1 is bidirectional temporal prediction, also called motion compensated interpolation. In bidirectional prediction mode two reference pictures are used, one in the past and one in the future. Macroblocks can be predicted using the past reference picture (forward prediction), the future reference picture (backward prediction), or an average of both reference pictures (interpolation). The pictures coded using bidirectional prediction are referred to as B-pictures.

The bit-stream is partitioned into slices. Each slice has its slice header that is used for resynchronization in case of bit-errors during transmission (see also Section 1.3.2). If a bit-error results in illegal data the damaged macroblocks are skipped and replaced by the latest correctly decoded macroblocks until the next slice is detected. Skipped macroblocks causes visible errors, which are small for P- and B-pictures but more obvious in I-pictures. This is improved in later versions of the MPEG standard.

1.2.3 MPEG-2 / H.262

After modifying and improving the MPEG-1 standard, ISO/IEC presented the well-known MPEG-2 standard (cf. [23] [29]), which uses the same coding techniques as MPEG-1. (The MPEG-2 standard has been adopted by ITU-T as standard H.262.) The main difference between MPEG-1 and MPEG-2 is that MPEG-2 uses a layered structure and supports a variety of different source input formats. Efforts have been made to make MPEG-2 have good support for interlaced video signals, digital TV and network communications. ISO/IEC planned to develop a third video coding standard, MPEG-3, to support high definition TV (HDTV). Instead, however, the HDTV work was fitted into the MPEG-2 standard at an early stage.

The MPEG-2 standard is optimized for bit-rates above 2 Mbits/s, and the layered structure allows for the creation of scalable systems. In a scalable system it is possible to decode a bit-stream to various quality levels according to the cost or complexity of the receiver. Another new feature in MPEG-2 is data partitioning. When using data partitioning, bit-stream data is reorganized in such a way that the compressed video signal becomes more robust to bit-errors (this is described in more detail in Section 1.3.2). Furthermore, error concealment is improved in MPEG-2. If a slice (see Section 1.2.2) is discovered to be erroneous, the error is concealed by replacing the slice with motion compensated pixels from the previous I- or P-picture.

1.2.4 H.263

H.263 [13] is an improvement of the preceding H.261 recommendation. It achieves significantly better image quality and manages to compress video to even lower bit-rates. An important reason for these improved results is that H.263 uses half pixel prediction when estimating motion. This means that, when matching the target macroblock with an equally sized block in the reference picture, we search half pixel displacements and are not restricted to integer pixel displacements. The image data in between pixels is calculated through interpolation. H.263 also has a more efficient VLC table for coding DCT-coefficients. In addition to the baseline H.263 coding algorithm, four coding options are included in Version 1 [13], which can be used together or separately. These are described below. Version 2 and Version 3 of the H.263 recommendation include several more coding options, which will not be discussed here.

Unrestricted Motion Vector mode

In this mode, motion vectors can be twice the normal length, and they are allowed to point outside the picture. When motion vectors point outside the picture, edge pixels are used as prediction for the non-existing pixels outside the picture.

Syntax-based Arithmetic Coding mode

This is an annex to H.263 that adopts the arithmetic coding technique. When using this mode, the reconstructed frame at the decoder will be the same as without the mode turned on, but fewer bits will be produced by the encoder. A static model containing probabilities for all the syntax elements (transform coefficients, motion vectors, etc.) resides in both encoder and decoder (compare Table 1.1). This syntax-based model is used in the encoder for arithmetic coding of the bit-stream and then the identical model is used in the decoder for arithmetic decoding of the bit-stream.

Advanced Prediction mode

Motion estimation, as described in Section 1.1.3, produces one motion vector per macroblock to represent the displacement of the block used for prediction. However, when advanced prediction is used, the encoder has the ability to decide whether to use one or four motion vectors per macroblock. Four motion vectors use more bits, but give better prediction. If four motion vectors are used, the macroblock is split into four 8x8 blocks, of which each is assigned one motion vector.

The advanced prediction mode also includes something called overlapped motion compensation [13], which will not be discussed here.

PB-frames mode

In this mode it is possible to encode a P-picture and a B-picture as one unit, hence the name PB-frame. The P-picture is predicted from the previously decoded P-picture and the B-picture is predicted from both the previously decoded P-picture and the P-picture currently being encoded. The latter may be a bidirectional prediction. In [13] it is claimed that this coding option can increase the picture-rate considerably without increasing the bit-rate much.

1.2.5 MPEG-4

MPEG-4 [11] [12] became an international standard in the beginning of 1999. The standard supports bit-rates typically between 5 kbits/s and 10 Mbits/s. With object based architecture, MPEG-4 introduces a completely new concept of video coding. MPEG-4 supports both natural and synthetic (computer generated) video, and data is represented using audio visual objects (AV-objects). A video object, i.e. the visual part of an AV-object, can be 2-dimensional or 3-dimensional and arbitrarily shaped, and represent for example a person's head.

The MPEG-4 standard is very flexible. It defines a syntactic description language, which is an extension of the C++ language, to describe the exact binary syntax for bit-streams.

Although MPEG-4 has many new features, the basic video coding techniques are similar to MPEG-2. It utilizes the same DCT based approach, but MPEG-4 also supports shape adaptive DCTs. Version 2 of the MPEG-4 standard, approved in December 1999, includes fully backward compatibility extensions.

1.2.6 H.26L

In September 1999, the first draft of the H.26L [14] proposal was available. H.26L is currently under development and it is supposed to replace its predecessor H.263. The H.26L functionality areas set up by ITU-T are as follows [17].

- High compression performance:
  - 50 % greater bit-rate savings from H.263 at all bit-rates.
- Simplifications "back to basics" approach:
  - simple and straightforward design using well-known building blocks.
- Flexible application to delay constraints appropriate to a variety of services:
  - low delay.
- Error resilience.
- Complexity scalability in encoder and decoder:
  - scalability between image quality and amount of encoder processing.
- Full specification of decoding (no mismatch).
- High quality application:
  - good quality also in high bit-rates.
- Network friendliness.

To fully satisfy all the above conditions simultaneously may be rather difficult. However, the core algorithms have been determined and the basic configuration of the algorithms is similar to H.263. An implementation (in the C language) was available along with the first proposal. As of today, H.26L is constantly changing and parts of the first draft are already modified and improved in later drafts. However, it is still interesting to display the main differences between the first draft H.26L codec and the H.263 recommendation. These are described below.

Only one regular VLC table used for symbol coding

Figure 1.4a shows the syntax of the codewords. Here, x_0, x_1, x_2, ... are the INFO-bits and take values 0 or 1. Figure 1.4b lists the first 9 codewords. Example: codeword number 4 contains the INFO 01. This design of the codewords is optimized for fast resynchronization in case of bit-errors. In this version of H.26L there is no arithmetic coding option, but it is believed that the match between coding model and the VLC will compensate for that [16].

$$\begin{array}{c}
1 \\
0\ x_0\ 1 \\
0\ x_1\ 0\ x_0\ 1 \\
0\ x_2\ 0\ x_1\ 0\ x_0\ 1 \\
0\ x_3\ 0\ x_2\ 0\ x_1\ 0\ x_0\ 1 \\
.\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .
\end{array}$$

Figure 1.4a Codeword syntax.

Codeword number	Codeword
0	1
1	0 0 1
2	0 1 1
3	0 0 0 0 1
4	0 0 0 1 1
5	0 1 0 0 1
6	0 1 0 1 1
7	0 0 0 0 0 0 1
8	0 0 0 0 0 1 1
:	: : : : : : :

Figure 1.4b The first 9 codewords.

One-third pixel positions used for motion prediction

In H.26L, one-third pixel positions are used for motion prediction (compare half pixel prediction in H.263). This has shown to have a great impact on image quality. Subjective image quality is especially improved. When estimating motion, the encoder also has the possibility of choosing a “stronger” interpolating filter. In some cases this filter gives a better prediction than the one-third pixel displacement.

Motion prediction using one-third pixel displacement is an example of a feature that has recently been improved and a method using one-sixth pixel displacements will be adopted into the standard.

Seven different block sizes used for motion prediction

There are 7 different block size-modes for motion prediction in H.26L. These include 1, 2, 4, 8 or 16 motion vectors per macroblock. A macroblock is partitioned into sub-blocks according to Figure 1.5.

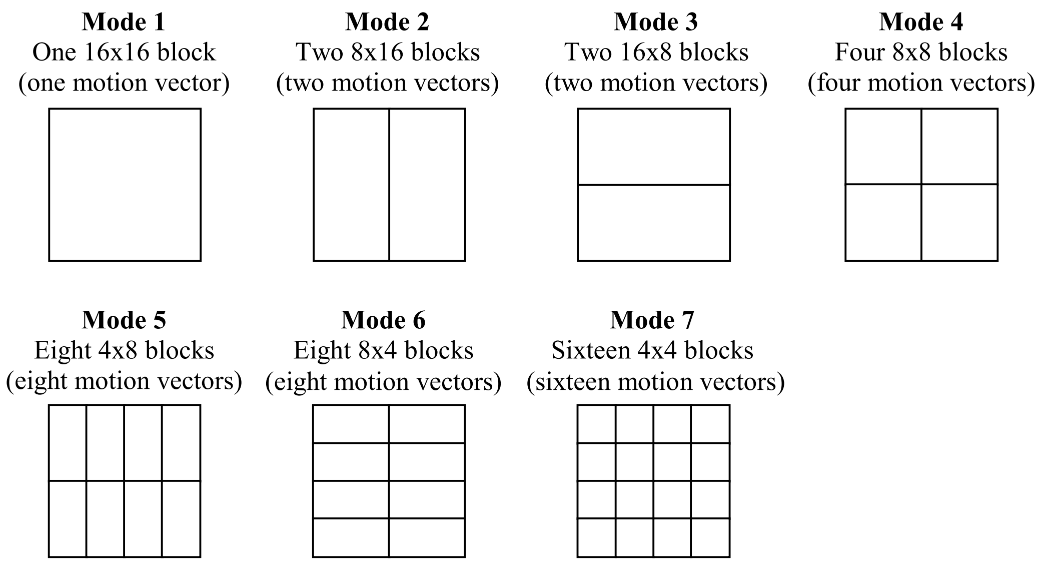

Figure 1.5 Partition of a macroblock for the 7 different block size-modes.

Integer transform applied on 4x4 blocks

The residual coding is based on an integer transform with essentially the same coding properties as the DCT. An advantage of this is that it allows **exactly** the same inverse transform in both coder and decoder. That is, there will be no "inverse transform mismatch." The transform is applied on 4x4 blocks, instead of 8x8 blocks, which is typical for most video coding standards. This results in fewer computations and less coding noise around moving edges [16].

Multiple reference frames may be used for motion prediction

There are no B-frames or PB-frames in H.26L. However, it is possible to allow the encoder to use more than one of the past pictures for motion estimation. If we choose to use multiple reference frames, for example two or three of the latest decoded frames, the encoder will choose the frame giving the best prediction for each target macroblock. The encoder then signals to the decoder for each macroblock which frame to use for the prediction.

Other features

Examples of other features in H.26L are intra prediction and double scan described below. In virtually all video coding techniques, frames are coded on a macroblock basis from left to right and from top to bottom. Furthermore, since the transform is applied on smaller blocks, the macroblocks are also coded from left to right and from top to bottom.

In H.26L, intra prediction is used when encoding INTRA-pictures. For each 4x4 block, except for edge-blocks that are treated differently, each pixel is predicted by a particular weighting of the closest 9 of the previously coded pixels. That is, the 9 pixels to the left and above the block. This exploits spatial redundancy between adjacent blocks and hence, a more efficient compression is achieved.

To describe the double scan feature of H.26L we need to take a step back and consider how the transform coefficients, or the symbols, are translated into the bit-stream. This is not done from left to right and from top to bottom, because that would be inefficient. When a block is transformed using a DCT-like transform, the resulting transform coefficients typically have larger values close to one corner of the block, and the coefficients close to the opposite corner are much smaller and often zero. In H.26L, data is organized so that transform coefficients with larger values are located in the top left corner of a block. For VLC and run-length coding purposes it is preferable to group symbols likely to be equal, and to end the “sequence of coefficients” with smaller coefficients, which are often zero. Therefore, a more efficient zig-zag scan (Figure 1.6a) is usually adopted.

Each sequence of coefficients is terminated by an end of block (EOB) symbol. In normal cases, the occurrence of the EOB symbol in the bit-stream typically matches that of the source model, which results in efficient compression. This is, however, not always true. For INTRA-pictures there are more transform coefficients that need to be coded, i.e. more non-zero coefficients, and the probabilities of the model are shifted so that there are too few EOB symbols in order to match the model well. This problem is solved by translating each 4x4 transform coefficient block into **two** sequences of coefficients. In this case, the coefficients are ordered according to the double scan method in Figure 1.6b.

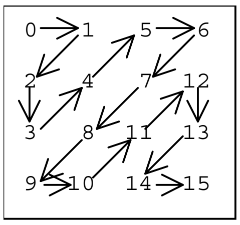

Figure 1.6a Single scan (zig-zag).

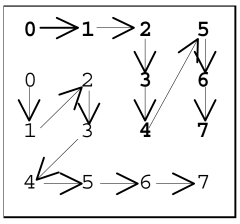

Figure 1.6b Double scan.

1.3 Areas of Research and Further Aspects of Video Coding

Motion estimation and residual coding are popular areas of research in video coding. How to make a video signal error resilient and robust are other examples of interesting research topics. Motion prediction can be improved by adopting more sophisticated models [19], and video coding using wavelet-transforms has been shown to give good results [6]. Occasionally, the lossless entropy coding stage in video coding seems to be overlooked, as Huffman and VLC techniques are almost taken for granted and more sophisticated methods are not mentioned. In Section 1.3.4, adaptive arithmetic coding is introduced, which often outperforms the better-known Huffman coding technique.

1.3.1 Rate-Distortion Theory

It is always possible to increase the compression if a decrease in image quality is allowed. Similarly, it is possible to obtain better image quality if we are less concerned about the bit-rate. This trade-off is rather obvious, however, it is not always an easy task to optimize the trade-off. The goal of rate-distortion theory [24] is to achieve highest possible compression, or lowest possible bit-rate, at a given image quality, or a given maximum distortion. This is done by allocating resources wisely.

For a given system and source it is possible to define a rate-distortion curve (R-D curve), Figure 1.7. The curve is obtained by plotting the distortion, i.e. a particular reproduction level at the decoder, versus the bit-rate based on specific parameters of the encoder. Parameters for controlling the bit-rate are for example quantization level, entropy coding (symbol occurrence), amount of and size of motion vectors, INTRA- versus INTER-coded compression, etc.

For each bit-rate, the best encoder/decoder pair is designed, and the achieved distortion is plotted against the bit-rate (Figure 1.7). The “best” points obtained in the set will define a convex hull. This is the boundary between achievable and non-achievable performance.

The obtained R-D curve can now be used to give the lowest possible bit-rate for a given distortion, but also to determine the optimum parameter settings, i.e., a point on the convex

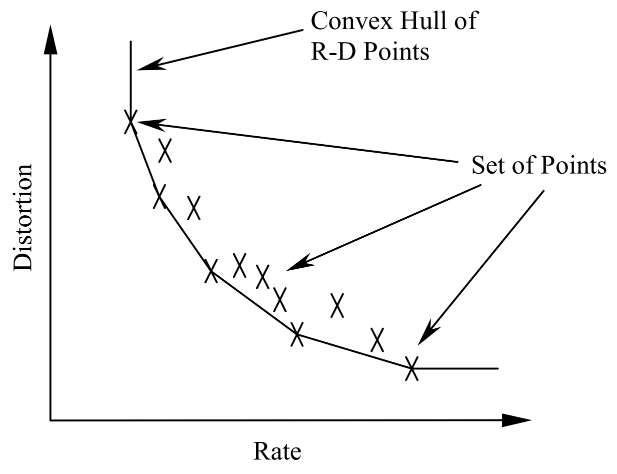

Figure 1.7 Rate-distortion curve. The “best” points in the set define the convex hull.

hull, for a given rate-distortion *ratio*. Two popular rate-distortion optimization techniques using an approach similar to “finding the best point on the R-D curve,” are Lagrangian optimization and dynamic programming [24].

In many cases, there are also one or more constraints imposed on the system. These include, for instance, maximum delay, maximum available memory for buffering, etc. and have to be considered when generating the rate-distortion curve.

The metric used for finding the minimum distortion is, almost exclusively, the MSE. It is undisputed that the MSE is not well correlated to human perception. It has been shown, however, that the difference in image quality between systems based on perceptually meaningful metrics and systems based on the MSE metric is less than expected, if not negligible [24].

An important step in rate-distortion optimization not discussed so far is the choice of model for the source. No matter how advanced a model, it can always be improved and made more sophisticated. It is, however, preferable to have a model that is not too complex. It should be simple, but still capture the main characteristics of the source. In many cases, a probabilistic model is chosen with parameters based on an appropriate training set.

1.3.2 Error Resilience

If a video sequence is compressed for the sole purpose of being **stored** more efficiently, for example on a computer, there is very little chance for bit-errors to occur in the data and hence, there is no need for the compressed video sequence to be error resilient. However, if we want to transmit the video sequence over, for instance, a computer network or a wireless communication channel it may be subjected to bit-errors. This does not call for error resilience if there are no limitations on the transmission time, and data can be retransmitted in case of bit-errors. On the other hand, if we are using video in, for example, real-time communication and it is important to have **low delay**, we will achieve best results by employing an error resilient video coding technique [28]. Furthermore, when the video signal is compressed, which is preferable because it reduces transmission time considerably, it also becomes more sensitive to bit-errors.

A typical method to make video codecs more error resilient is to employ forward error correction (FEC). The protection is introduced in the encoder and used by the decoder to correct errors in the bit-stream. Unfortunately, this also increases the bit-rate. In addition to FEC, syntactic and semantic error detection techniques can be applied at the decoder. In a block-based video compression technique using DCT and VLC codes, the following checks can be made to detect errors in the bit-stream:

- An invalid motion vector is found.
- An invalid VLC table entry is found.
- A DCT coefficient is out of range.
- The number of DCT coefficients in a block exceeds N^2 (where the DCT is applied on NxN blocks).

When the decoder detects an error at a certain location in the incoming bit-stream, it is unlikely that the error actually occurred at the same location. It is, in fact, impossible for the decoder to pinpoint the exact position of an error in the bit-stream. The reason is that the decoder looses synchronization with the encoder when an error occurs. In order to make it possible for the decoder to resynchronize with the encoder, unique codewords called resynchronization markers, or resync markers are put into the bit-stream at various locations. This allows the decoder to, at least, isolate the error to be in between two resync markers.

While resynchronization techniques are adopted by virtually every video compression system, there exist other useful but not always employed error resilience tools, some of which are described below [28].

Reversible Variable Length Codes

Assume the decoder detects an error in between two resync markers. There is actually a technique that allows the decoder to isolate the error to a narrower interval than the interval between two resync markers. The idea is to decode the bit-stream in between the resync markers both forwards and backwards. When the decoder detects an error, we know that an error occurred in the bit-stream somewhere between the resync marker (where the decoding began) and where the error was detected. When decoding is done in both directions, the error can be isolated to have occurred in between the two detected errors.

In order to make backward decoding possible we employ reversible variable length codes (RVLCs), which have the unique prefix property both forwards and backwards. This means that no codeword exists within another codeword as the leading or ending 1s and 0s. Hence, when a codeword is read from the front as well as from the back it will be uniquely determined and not confused with another codeword. One way to construct RVLCs is by adding a fixed prefix and suffix to constant Hamming weight VLCs, i.e. VLCs with the same number of 1s.

The RVLC technique allows us to use some of the non-corrupted data in between resync markers, instead of discarding it all. However, the technique described above needs to be improved and becomes more complex when there is more than one bit-error in between resync markers.

Data Partitioning

The resync marker, plus all the data up to the next resync marker, is referred to as a video segment, or a slice (Figure 1.8). A video segment contains a number of macroblocks, and hence the information in the video segment consists of DCT coefficients and one or more motion vectors (MVs). A few bits are also needed for quantization information, type of

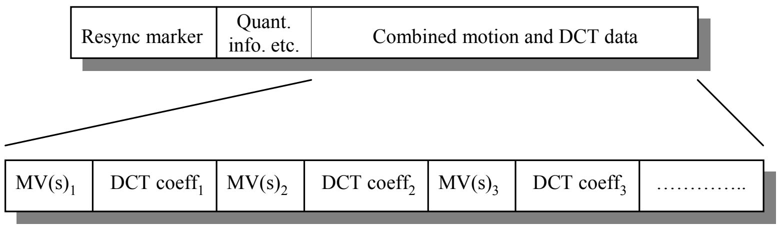

Figure 1.8 Organization of the data within a video segment.

macroblock encoding, etc. When motion and DCT data are coded together as in Figure 1.8 (the subscript indicates the macroblock number), typical video decoders will, if an error is detected, discard all the macroblocks in the video segment and replace them with the corresponding macroblocks in the previous frame.

When utilizing the data partitioning technique, the data is partitioned into motion data and DCT data as in Figure 1.9, and a unique marker, in MPEG-4 called a motion boundary marker, separates the motion data from the DCT data. Now, if the decoder detects an error, it

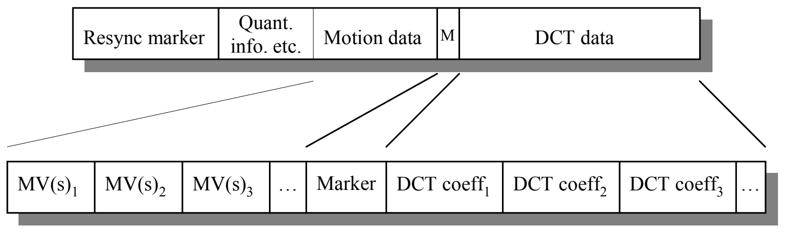

Figure 1.9 Organization of the data within a partitioned video segment.

checks if the unique marker was correctly decoded, and if so the decoder discards only the DCT data in the video segment and uses the motion data to conceal the error more efficiently than when using information only from the previously decoded frame.

The data partitioning technique is based on the assumption that motion vectors account for a smaller part of the total amount of data in a segment than the DCT data, and thus an error is more likely to occur in the DCT data part. Since this is true for virtually every video compression standard, data partitioning is a good and widely used error resilience technique.

Header Extension Code

In typical video compression techniques, a header containing important information about, for example, the video format and the type of encoding (INTRA or INTER), is transmitted along with each video frame. If this information is corrupted, the decoder has no choice but to discard the entire frame and use the previously decoded frame instead. However, when employing the HEC, the important information is repeated within the frame, so that if the frame header is corrupted it is still possible to decode the rest of the frame.

1.3.3 Advanced Motion Estimation

There are several ways to improve motion estimation as described in Section 1.1.3. More than one reference picture can be used (H.263, H.26L, MPEG), both past and future reference pictures can be considered (MPEG), the search for the best matching block can be made faster, and the models can be made more sophisticated, capturing motion better.

Gradient Matching

In the block matching technique described in Section 1.1.3, an exhaustive search is done. This means that the best matching block is found by comparing **all** the possible displacements of the block within a specific search area. In other words, all possible motion vectors are tried. Many video coders employ this technique because it guarantees that the best matching block is found and thus highest possible compression is achieved. There are, however, other ways of tracking motion, some of which are considerably faster.

In gradient matching, instead of doing an exhaustive search, we try to find the best matching block, or an equally good prediction, by an intelligent search of only a few blocks. This is done by first choosing a *starting* block and then trying the motion vectors corresponding to the smallest possible displacement in typically 8 directions away from the starting block. When the best matching block is found among these 9 blocks, the same procedure is applied around the best matching block, and so on. This is repeated until no better matching block is found in any direction and hence, the minimum is found, and the corresponding block will be used for prediction. The starting block is typically the macroblock in the previous frame corresponding to the target macroblock. Or, the block found using the motion vector for the macroblock in the previous frame corresponding to the target macroblock (motion vector prediction).

The advantage of this block matching method is that the complexity is reduced considerably. A drawback is that it is possible that the minimum is local and hence not the overall best match, which will give a small decrease in coding efficiency. This decrease is in most cases negligible, however, compared to the significant gain in encoding speed.

Motion Models and Regions of Support

In [19] a number of different motion models are introduced, starting with a simple translational model and adding parameters step by step to achieve far more advanced and sophisticated models capturing complex movements.

In a video sequence, 3D motion is represented by spatially constant 2D motion through orthographic projection. This 2D motion is called apparent motion or optical flow and is recovered from the intensity of the video sequence. In typical video coding a simple translational, i.e. two parameters, model is used which accounts for a rigid 3D translation. If we want to capture 3D affine motion (translation, rotation or uniform stretching), more sophisticated models with 6, 8 or even up to 12 parameters (quadratic model) needs to be adopted. In general, models using more parameters, and thus having higher function orders, describe motions more precisely. However, it comes with the disadvantage of increased coding cost, sometimes considerably.

How precisely a model describes motion also depends on the region of support for the model, i.e., the pixels to which the model applies. Since it is very difficult to capture the *true* motion in a video sequence, the smaller the region of support, the better the approximation. Four main regions of support are [19]:

- **Global**. The region of support is the entire image and all pixels are displaced in a uniform manner. This is the most constrained case, because very few parameters describe the motion of all pixels. Global motion is typically caused by a camera pan.

- **Dense**. This corresponds to motion of individual pixels, and is the least constrained case. A dense region of support captures motion very well, but at the same time this method is the most complex due to the high number of parameters involved.
- **Block-based**. A compromise of the two extremes above. Images are partitioned into non-overlapping rectangular regions (compare macroblocks). This is used in typical video coders together with a translational model. The combination is simple and computationally efficient, but does not capture general motion such as rotation, zoom and deformation.
- **Region-based**. In this considerably more sophisticated method the partitioning of the image is strongly correlated to the motions. All pixels in a region arise from objects that move together.

Advanced motion models and intelligently chosen regions of support are useful not only in video processing and computer vision but also when compressing video. Although older standards (H.261, MPEG-1 & 2) use simple methods like block-based translational models, newer standards, such as MPEG-4, have adopted more sophisticated techniques using models with region-based and global regions of support. Yet another way of improving motion estimation is to use the block-based method with variable-size blocks. In this approach, which is employed in H.263, the block size is reduced wherever a smaller block size improves motion compensation.

1.3.4 Adaptive Arithmetic Coding

Many techniques can be improved by allowing them to adapt to local statistics (i.e. spatially or temporally local). The adaptation typically adds complexity, but in many cases it also results in a more efficient compression. Since entropy coding in most video coding standards is easily separated from the rest of the coding algorithms, making this stage adaptive does not introduce a dramatic change to the video codec. As mentioned earlier, many video coders of today use VLCs, or Huffman codes, for entropy coding. It is possible to combine these with adaptive models, however, it usually results in complex and inelegant solutions with little gain in compression efficiency [3]. Arithmetic coding, on the other hand, is relatively easy to combine with adaptive models and the resulting gain in compression efficiency is for many sources significant. Adaptive models reach an entropy closer to that of the source than do non-adaptive models, and arithmetic coding has the ability to represent the probabilities produced by the model exactly. This makes adaptive arithmetic coding (cf. [3] [8]) a powerful tool in both video compression [6] and image compression [7].

The nature of the adaptation for arithmetic coding can vary. However, the most straightforward method is to update an array of frequency counts every time a symbol is encoded/decoded (this is done *on the fly*). Assume that we have an alphabet (in video coding, transform coefficients and motion vectors etc.) of 100 symbols. Each symbol has a corresponding slot in a frequency array where the occurrence of the symbol is stored. Initially the frequency array contains 1s for all the symbols, meaning that each symbol has a probability of 1/100. When a symbol is encoded the count in the frequency array is incremented by one and the symbol will have a probability of 2/101, and so on. An identical procedure in the decoder ensures that both encoder and decoder always have exactly the same model. Hence, no models or probability tables have to be sent from the encoder to the decoder and no initial histograms, or probability measures, have to be generated.

A drawback of the traditional implementation of adaptation described above, is that the number of operations increase linearly with the number of symbols. This can, however, be improved. In [4] an adaptation technique using hierarchical sums is presented. In this approach the computational expense increases only logarithmically with the number of symbols.

A slightly different approach when exploiting the advantages of adaptation is to make use of different contexts [20] [25] [27]. In context-based adaptive arithmetic coding, statistics of the source are used not only to "update probabilities," but also to choose a context for which the arithmetic coding will be based. A particular conditional probability distribution corresponds usually to each context type. In for example INTRA-coded compression, a context can be formed from pixel values in the vicinity of the pixel being encoded [25], or from the prediction error of those pixels [20]. Another approach is to let transform coefficients with similar properties, typically magnitude, form different contexts [27].

Combinations of adaptive arithmetic coding with other techniques have been proposed. In [21] the adaptive arithmetic coding is taken to higher dimensions by the use of vectors, or joint entropy coding. Here, symbols are grouped into vectors of different dimensions, and each vector dimension has its corresponding table in which the vector is stored. These tables are generated on the fly. When a vector, i.e. a group of symbols, is encoded/decoded it is either looked up in a table, partitioned into a lower dimension and looked up in a table containing lower dimensional vectors, or encoded/decoded traditionally symbol-by-symbol. In either case, the symbols are more efficiently coded or the model is adapted to the source, providing for more efficient coding in the future. The obvious drawback of this method is the increase in complexity and memory.

The behavior of the adaptive arithmetic coding technique in error-prone environments is a concern. When an adaptive arithmetic coded bit-stream is subjected to bit-errors, every bit-error can have serious consequences for the ability to decode the bit-stream. If a bit-error results in an incorrectly decoded symbol, this does not only mean that the correct symbol will not be decoded but also that the adaptive part is damaged. In a system where the symbol frequencies are counted and a model is built up on the fly, the encoding of each symbol is dependent on the frequency of the already decoded symbols. In such a system, a bit-error usually results in that the model has to be reinitialized and if the location of the bit-error is unknown, typically all the data has to be discarded. It is indeed natural to suspect that error resilience and adaptive arithmetic coding is difficult to combine in a successful way.

In [2] it is claimed that for a very low bit-rate environment a computationally simple method using static Huffman coding is competitive to adaptive arithmetic methods. It is clear that adaptive arithmetic coding outperforms VLCs and Huffman coding in theory. In many cases, this is also true for practical applications.

1.4 Purpose and Goal

The purpose of the thesis presented in this report is to investigate the possibilities to make video compression and the H.26L video codec in particular, more efficient by means of adaptive arithmetic coding.

An initial video codec evaluation in Section 2 gives a general idea of how much can be gained by improving the entropy coding in H.26L. It is difficult, however, to investigate how much could truly be gained through adaptive arithmetic coding without implementing such a scheme. Therefore, the goal will be to integrate the adaptive arithmetic coding technique with the H.26L video codec and examine the gains (or losses) in compression efficiency.

1.5 Outline

The remaining part of this report is divided into four sections.

The results of a video codec evaluation are presented in Section 2. The H.26L video codec is tested against the H.263 recommendation and the efficiency of the arithmetic coding in H.263 is examined. The H.26L syntax is presented allowing an entropy study of H.26L and an investigation of the bit distribution for different data types, or syntax elements.

Section 3 begins with a presentation of three existing adaptive methods. Then follows a description of the functionality of the adaptive arithmetic codec. Here is described how the adaptation is carried out and which parameters can be set for the adaptive arithmetic codec.

In Section 4 three experimental models for adapting to local statistics are described in detail. Each model is evaluated using one or more video sequences and the corresponding bit-rate reductions are presented. The third and last model achieves the best results, which are presented in the end of the section. Section 4 also contains a general discussion about model design.

Conclusions can be found in Section 5. Issues such as error resilience and rate-distortion optimization in combination with adaptive arithmetic coding are discussed. The document ends with ideas for future work and guidelines for a fourth model are presented.

2. Video Codec Evaluation

The H.26L video codec is built on similar techniques as traditional video codecs. There are many differences between H.26L and older video codecs however. One important difference is the more sophisticated motion estimation technique. The improved motion estimation is most probably the main reason for the improved performance of H.26L. We will see in Section 2.4 that a typical H.26L bit-stream contains a relatively high percentage of data used for motion estimation.

In this section, video codecs are evaluated by encoding and decoding different video sequences. The results differ but are in many cases very similar. Hence, results for only a couple of video sequences are presented. When comparing two video codecs against each other, it is obvious that the fairest procedure would be to encode and decode a series of video sequences and, either present all the results separately, or average all the results together. To encode and decode a whole series of video sequences is, however, very time consuming and not necessary to get a general idea of performance differences. Furthermore, when certain aspects of a video codec are to be studied, averaging makes it difficult to examine why and for which parameters the differences occur.

2.1 H.26L versus H.263

To investigate the improvements of the H.26L video codec, it has been tested against its predecessor H.263. Such a comparison is most evident since the performance goal for H.26L is to get 50% greater bit-rate savings from H.263. Furthermore, the primary objectives for H.26L and H.263 are similar and both are optimized for low bit-rates.

2.1.1 The SNR Metric

The signal to noise ratio (SNR) [23] is the most commonly used metric for image quality in image and video compression. The signal is an image or a video sequence passing through a compression system.

When an image is compressed and then reconstructed, a measure of the noise in the reconstructed image is obtained as the MSE [23]

$$MSE = \frac{1}{M \times N} \sum_{m=1}^{M} \sum_{n=1}^{N} \left(x[m,n] - x_r[m,n] \right)^2 ,$$

where x is the original image, x_r is the reconstructed image, M and N is the width and the height of the image respectively, and *x[m,n]* represents the pixel in line m and column n.

The SNR of the image is then defined as

$$SNR_{IM} = 10 \log_{10} \frac{255^2}{MSE} .$$

The values of the pixels lie in between 0 and 255, however in the calculation of SNR_{IM} the pixel values are divided by 255^2, and thus, normalized to lie in between 0 and 1.

The SNR of a sequence of images is defined as the average SNR over all images. The SNR for a reconstructed frame in a video sequence is calculated using the corresponding frame in the original video sequence. Thus, to calculate the SNR of a video sequence, first the SNR_{IM} for each *pair* of frames in the reconstructed video and the original sequences is calculated, and then the average of all SNR_{IM} is calculated as

$$SNR_{VS} = \frac{1}{F}\sum_{f=1}^{F} SNR_{IM}(f),$$

where F is the number of frames in the video sequence and $SNR_{IM}(f)$ is the SNR for the f:th image.

When comparing the difference between two SNR curves, it is important to remember that SNR is a logarithmic function. On e should also keep in mind that SNR is a quantitative metric which can not be directly transferred into a qualitative human perception metric.

2.1.2 Video Sequences and Formats

Two of the sequences used in the evaluation are Foreman (Figure 2.1a) and Silent (Figure 2.1b). Foreman is a video sequence of a foreman and a construction area filmed with a handheld camera and thus includes irregular motion of both global and local nature. Silent is a video sequence of a person doing sign language. It is filmed with a steady camera and visible motion arises from the person only. Therefore, for a given SNR value, Silent can be compressed to a lower bit-rate than Foreman.

The current version of H.26L supports two different video formats: common intermediate format (CIF) and quarter common intermediate format (QCIF). The CIF format has 352 pixels/line and 288 lines/frame while the QCIF format is a quarter of the size of the CIF format and has 176 pixels/line and 144 lines/frame.

Figure 2.1a Foreman.

Figure 2.1b Silent.

2.1.3 Test Issues and Procedure

When compressing a video sequence, the bit-rate and thus the achieved SNR is controlled by altering the quantization step. In H.26L, as well as in H.263, quantization steps can be set between 0 and 31. The higher the quantization step, the fewer quantization levels will be used and the fewer bits are used to represent the data. H.26L may handle different quantization steps for each frame; only two different quantization steps are used in the tests, one for the first frame and another for the remaining frames. The first frame is necessarily INTRA-coded and the remaining frames are (in all the tests presented in this document) INTER-coded. This means that one quantization step is set for the INTRA-picture and another quantization step is set for the INTER-frames.

The fact that the quantization step is set separately for the first and the remaining frames is important when calculating the average SNR for entire sequences. If the video sequence to be coded is long, varying the quantization step for the first frame, and thus also varying the number of bits for the first frame makes a negligible difference in total bit-rate. However, the quality of the following frames depends on the quality of the first frame. This is especially true for video sequences containing little motion. Thus, lowering the quantization step for the first frame, but keeping the quantization step for the remaining frames the same will improve the SNR values for the remaining frames as well. Hence, there is a trade-off between the percentage of bits in the first frame and the overall SNR.

The two video codecs are run 13 times each in each test. Different quantization steps in each run give different bit-rates and the corresponding SNR values are calculated. This results in 13 points in an “SNR vs. bit-rate”-graph for each video codec. The curves are generated through linear interpolation. To reflect the difference in SNR between the two video codecs best, the quantization steps are chosen so that 13 *pairs* of points are generated, where the two encodings in each pair have approximately the same bit-rate. The 13 pairs of points span a region in SNR from about 27 to about 45 dB, and in bit-rate from about 25 kbits/sec to about 1 Mbits/sec, where the exact figures of course depends on the video sequence. The 13 points in each test run correspond, for the H.26L codec, to the quantization steps in Table 2.1.

Table 2.1 The 13 quantization steps for INTER-frames for the H.26L test runs (included are also bit-rates and compression ratios for the QCIF format of the Foreman video sequence).

Point	Q. step (INTER)	Bit-rate (Foreman)	Comp. ratio (Foreman)
1	28	37 kbits/sec	1:247
2	27	42 kbits/sec	1:217
3	26	47 kbits/sec	1:194
4	24	58 kbits/sec	1:157
5	22	74 kbits/sec	1:123
6	20	96 kbits/sec	1:95
7	19	108 kbits/sec	1:84
8	18	126 kbits/sec	1:72
9	17	142 kbits/sec	1:64
10	16	166 kbits/sec	1:55
11	12	302 kbits/sec	1:30
12	8	539 kbits/sec	1:17
13	4	942 kbits/sec	1:10

Included in the table are also corresponding bit-rates and compression ratios for the QCIF format of the Foreman sequence. There are more points for lower and intermediate bit-rates to account for the greater derivative in this range. Furthermore, higher bit-rates in the range around 1 Mbits/sec represent, for the QCIF format, a compression of approximately 1:5 to 1:15, and in general applications require a stronger compression. The high bit-rate parts of the graphs are less interesting also because neither H.26L nor H.263 is optimized for high bit-rates.

2.1.4 Example of Trade-Off Between Image Quality and Quantization Step

In order to get a feel for how great an impact the quantization step for INTER-pictures has on image quality, three encoded and decoded images are presented below. Figures 2.2a-c show frame number 46 of the Foreman video sequence in the QCIF format with INTER-quantization steps 4, 19 and 28, respectively. We see that the difference between the image

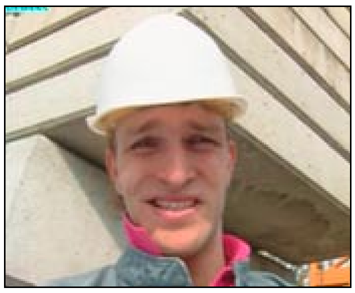
Figure 2.2a Q. step 4.

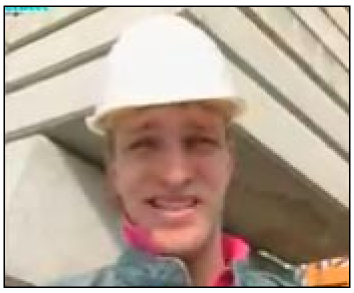
Figure 2.2b Q. step 19.

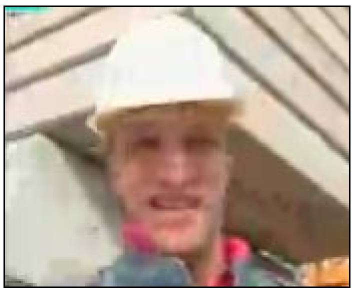
Figure 2.2c Q. step 28.

qualities is quite obvious. For a quantization step as low as 4 (Figure 2.2a), it is difficult to tell the difference in image quality from the uncompressed video sequence (compare Figure 2.1a). When the quantization step goes above 19 (Figure 2.2b), the degradation in image quality becomes rather obvious and irritating. For quantization step 28 (Figure 2.2c) the image quality is too poor even to tell who is in the picture.

2.1.5 Test Specifications

300 frames of the Foreman and Silent video sequences in the QCIF format have been encoded and decoded in each run. Both sequences have a frame-rate of 30 frames/sec, thus, 300 frames correspond to 10 seconds of video.

In order to make a fair comparison between the H.26L codec and the H.263 codec, the quantization step for the first frame is, for both encoders, set so that the SNR value for the first frame becomes approximately equal to the average SNR value for all frames.

Version 1 of the H.263 codec has been used, and the unrestricted motion vector, syntax-based arithmetic coding and advanced prediction coding options were turned on.

2.1.6 SNR vs. Bit-Rate for Foreman and Silent

In Figure 2.3 and 2.4 below are the results for Foreman and Silent, respectively. It is clear that for very low bit-rates the difference between H.263 and the current version of H.26L is very small. The improvements in SNR for H.26L over H.263 are starting to show between 80 to 100 kbits/sec. The slope around 220 kbits/sec is, in both figures, maintained also for higher bit-rates, and the relative improvement in SNR is approximately constant with bit-rates increasing over 100 kbits/sec.

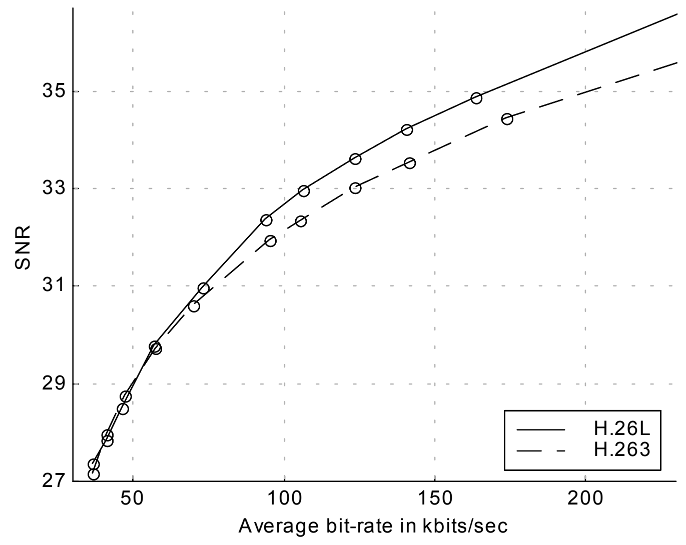

Figure 2.3 SNR vs. bit-rate for H.263 and H.26L (Foreman, QCIF).

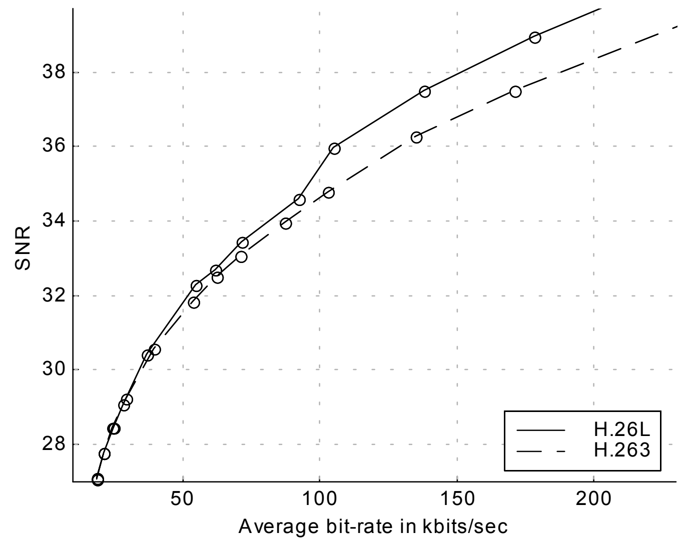

Figure 2.4 SNR vs. bit-rate for H.263 and H.26L (Silent, QCIF).

2.2 Arithmetic Coding versus VLCs for H.263

As mentioned in Section 1.2.4, there is a coding option in H.263 that employs arithmetic coding. To get a general idea of how much the bit-rate is reduced when this coding option is used, the H.263 codec with arithmetic coding is tested against the H.263 codec without arithmetic coding. When arithmetic coding is not employed, the default VLC technique is adopted.

2.2.1 Test Specifications

300 frames of the Foreman video sequence in the QCIF format have been encoded and decoded in each run. For each particular choice of quantization level the H.263 codec has been run twice, once using VLCs and once using arithmetic coding. Since the syntax-based arithmetic coding option only affects the entropy coding part of the video coding process, the decoded sequences are identical for a given quantization step, whether or not arithmetic coding is used. Hence, the test reflects exclusively the difference in bit-rate due to arithmetic coding.

2.2.2 Bit-Rate Reduction Using Arithmetic Coding

Figure 2.5 shows the bit-rate reduction in percent for increasing bit-rates achieved by the arithmetic coding option in H.263. The slope around 400 kbits/sec is maintained for higher bit-rates.

What the syntax-based arithmetic coding option essentially accomplishes is a more exact representation of the probabilities in the entropy coding model (refer to Section 1.1.4). Therefore, Figure 2.5 also visualizes the model mismatch for the VLC approach for different bit-rates. The more that is gained using arithmetic coding the worse is the match of the VLC

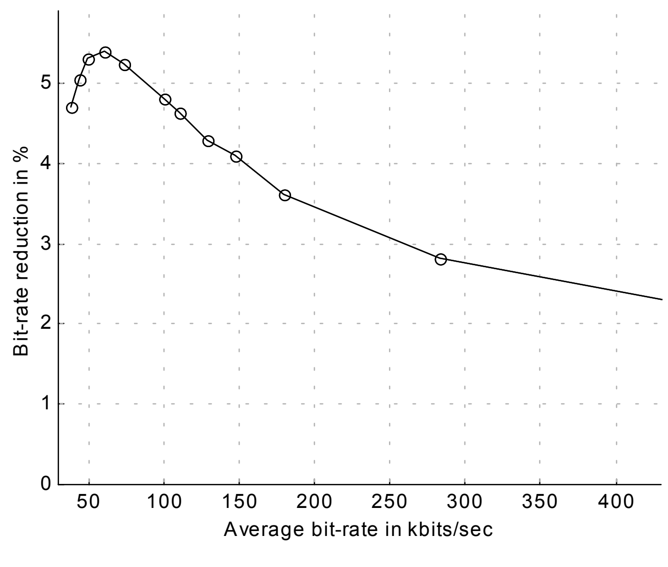

Figure 2.5 Bit-rate reduction for H.263 using arithmetic coding (Foreman, QCIF).

tables. As seen in Figure 2.5 the VLC tables are well matched for higher bit-rates, and here we do not gain much by using arithmetic coding. For low bit-rates, however, the compression can be made more than 5% more efficient.

2.3 The H.26L Syntax

An understanding of the H.26L syntax is necessary to investigate the entropy of an H.26L bit-stream. The first version of H.26L uses 10 different data types, or syntax elements in the bit-stream.

In video coding in general, the data types are often separated into transform coefficient data, motion vector data and mode data, where the number of bits used for mode data usually is only a small fraction of the number of bits used for other data.

2.3.1 Picture Sync (Psync)

The first codeword for each picture in an H.26L video sequence is the Psync. It is also used to indicate the end of the entire sequence. Besides working as a picture synchronization, Psync also contains a picture sequence number, which quantization step to be used, the picture format (QCIF or CIF), and whether it is a picture header or an end of sequence (EOS) symbol.

2.3.2 Picture Type (Ptype)

This codeword indicates which out of three types each picture is:

1. INTER-picture with prediction from the last decoded frame.
2. INTER-picture with prediction from frames further back in the sequence than the last decoded frame.
3. INTRA-picture.

2.3.3 Macroblock Type (MB_Type)

The MB_Type codeword is the first codeword in each macroblock in INTER-pictures. It tells us in which mode the macroblock is coded. There are nine possible modes of which mode 1 to 7, described in Section 1.2.6, defines the partition of macroblocks and how many motion vectors are used for each macroblock. The two additional modes, mode 0 and 8, represent skipped macroblock and INTRA-coded macroblock, respectively.

2.3.4 Intra Prediction Mode (Intra_pred_mode)

There are five different intra prediction modes that indicate how 4x4 sub-blocks in INTRA-coded macroblocks are to be predicted. Each mode corresponds to a particular weighting of different pixels in adjacent blocks (see Section 1.2.6). The modes to be used are indicated using eight Intra_pred_mode codewords for each INTRA-coded macroblock.

2.3.5 Reference Frame (Ref_frame)

Ref_frame codewords are used only if Ptype indicate possibility to predict macroblocks from frames other than the last decoded frame. In this case a Ref_frame codeword is sent for each macroblock, containing information about which past frame is to be used for prediction.

In this document, macroblocks have been predicted only from the last decoded frame in all tests of H.26L, and hence, no Ref_frame codeword occurs in any bit-stream.

2.3.6 Motion Vector Data (MVD)

For each macroblock, except INTRA-coded and skipped, 1, 2, 4, 8 or 16 motion vectors are sent. A motion vector for a particular block is always predicted from motion vectors for adjacent blocks. The motion vector is then represented by two MVD codewords, one MVD codeword for the horizontal component, and one MVD codeword for the vertical component of the motion vector. These MVD codewords represent the difference between the motion vector components to be used and the prediction of the motion vector components.

2.3.7 Transform Coefficients (Tcoeff_luma, Tcoeff_chroma_DC & Tcoeff_chroma_AC)

Transform coefficients in H.26L are coded using run-length coding (see Section 1.1.4). Each 4x4 block is zig-zag scanned (see Section 1.2.6) resulting in a sequence of coefficients. A transform coefficient codeword corresponds to the values *RUN* and *LEVEL*. These two values represent the number of zero coefficients preceding a non-zero coefficient and the level of that non-zero coefficient, respectively. The sequence of coefficients becomes a sequence of run-level-pairs. Each sequence of run-level-pairs is terminated by an end of block (EOB) symbol. When the decoder sees this symbol it knows that all the run-level-pairs in the current block have been decoded and that the following run-level-pairs belong to the next block. The ending zero coefficients in a sequence of coefficients do not need to be encoded since the decoder assumes any remaining coefficients to be zero when it receives the EOB symbol.

The transform coefficients are divided into luma transform coefficients (Tcoeff_luma) and chroma transform coefficients. Luma transform coefficients contain information about the luminosity of the pixels in the picture, i.e. a grayscale from black to white, where no intensity represents black and maximum intensity represents white. Chroma transform coefficients contain the additional information needed for the pixels to give color to the picture. When the luminosity of the picture is known, only two more components are needed to represent the color in the picture. This particular coding of luminosity and color information is known as

YUV coding, where Y is the luminosity, U is the intensity of the red color subtracted by the luminosity and V is the intensity of the blue color subtracted by the luminosity. The human eye is more sensitive to luminosity than to U and V, and the separation into Y, U and V components allow for a sub-sampling of the U and V components. This results in a more efficient coding since the importance of different parts of the data is weighted in a way similar to the filtering process taking place in the human eye. In a video sequence of the YUV format, the Y information accounts for twice as much of the total amount of data as the U and V information together.

The chroma transform coefficients are separated into DC coefficients (Tcoeff_chroma_DC) and AC coefficients (Tcoeff_chroma_AC). In a transformed sub-block the DC coefficient, which is the most important coefficient, is positioned in the top left corner. The rest of the coefficients are called AC coefficients. The reason for the separation is that the DC coefficients are subsequently transformed by a different transform. This gives a new, and more efficient, representation of the DC coefficients, which naturally also needs a specific VLC table.

2.3.8 Coded Block Pattern (CBP)

A CBP codeword is sent prior to the transform coefficient codewords for each non-skipped macroblock. CBP codewords are used to indicate which sub-blocks in a macroblock contain non-zero coefficients. With this approach it is not necessary to send EOB symbols for all sub-blocks with only zero coefficients. That is, less EOB symbols need to be sent and better compression is achieved. A CBP codeword can be viewed as a global EOB symbol for several sub-blocks. If many or all the 4x4 sub-blocks in a macroblock contain non-zero coefficients, CBP codewords results in a less efficient coding. This is because sub-blocks containing non-zero coefficients are always terminated by an EOB symbol making the CBP information obsolete. However, since in general few 4x4 sub-blocks contain non-zero coefficients, using CBP codewords usually gives a better compression, especially for low bit-rates.

2.4 Bit Distribution for Different Data Types

In order to compress the H.26L bit-stream efficiently it is useful to have knowledge about the distribution of bits over different data types. That is how many percent of the bit-rate is used for transform coefficients, motion vectors, mode data, etc. In general there are more motion vectors and less transform coefficients for low bit-rates and more transform coefficients and less motion vectors for high bit-rates. The distribution achieved after encoding is video sequence dependent as well, but it is negligible compared to the bit-rate dependency.

2.4.1 Test Specification

The QCIF format of the Foreman sequence has been encoded using the quantization steps in Table 2.1. Statistics for all different data types were collected during each run, in which 300 frames were encoded and decoded. Note that when as many as 300 frames are coded and only the first frame is INTRA-coded, data types used only for INTRA-coded macroblocks, such as the Intra_pred_mode data type, give a negligible contribution to the overall bit-rate.

2.4.2 Bit Distribution for Different Data Types in H.26L

In Figure 2.6 the bit distribution with respect to bit-rate is presented. This is a typical bit distribution for H.26L. Each curve shows the percentage of bits used for the corresponding data type. The "Coeff data" curve represents all the transform coefficients, i.e. Tcoeff_luma, Tcoeff_chroma_DC and Tcoeff_chroma_AC codewords. The "MV data" curve represents motion vector data only. The most bit-consuming data type next after transform coefficients and motion vector data is CBP, which is represented by the "CBP data" curve. To give a clear presentation of the distribution, no other data types have been represented alone. The "Mode & Sync" curve represents all the data types not represented by other curves.

The slopes around 220 kbits/sec are maintained also for higher bit-rates. H.26L, on average, uses a fair amount of bits for motion vectors. This is most surely due to the possibility to use as many as 16 motion vectors per macroblock.

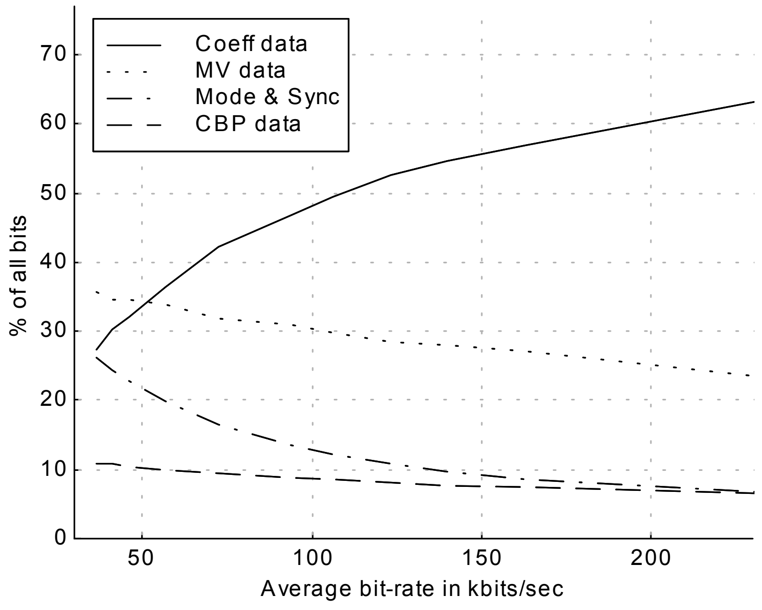

Figure 2.6 Distribution of bits over different data types in H.26L (Foreman, QCIF)

2.4.3 A comparison between H.26L and H.263

The same test was performed also for the H.263 codec to visualize the difference in bit distribution between H.26L and another major video coding standard. Figure 2.7 shows the bit distribution for H.26L and H.263 for transform coefficients and motion vector data. It is

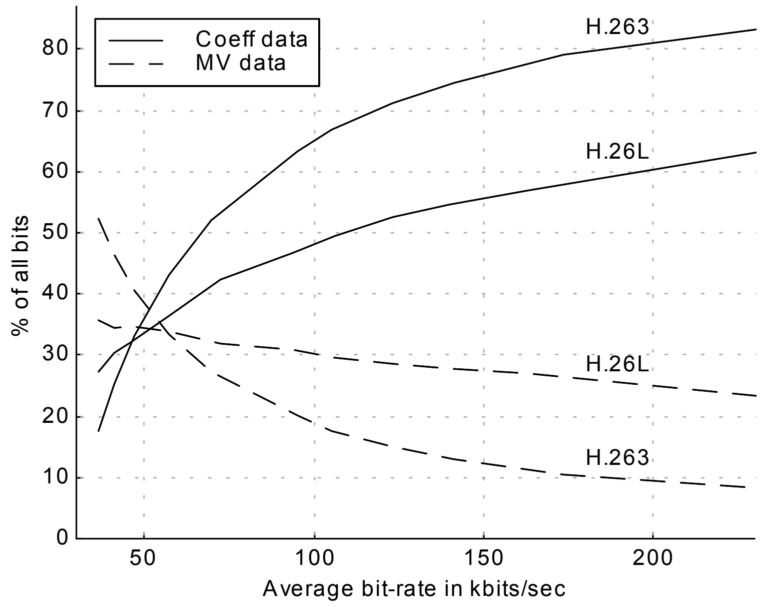

Figure 2.7 Bit distribution for different data types in H.26L and H.263 (Foreman, QCIF).

clear that H.26L has a more even distribution for all bit-rates, while H.263 has a distribution that is probably more similar to traditional video coding standards.

The slopes around 220 kbits/sec are maintained also for higher bit-rates. Implicitly shown in Figure 2.7 is also the fact that, regardless of bit-rate, H.26L uses considerably more bits for mode and sync codewords, than does H.263.

2.5 Entropy Study of H.26L

An investigation of the entropy of a typical H.26L bit-stream is of interest. That gives us information both about the performance of the codec and the possibilities for improvement. By going deeper into the composition of the H.26L bit-stream it is possible to study the entropy of different building blocks or syntax elements as well. This will help us further in finding a more efficient entropy coding technique. But first of all, let us review the modeling and entropy coding in the H.26L video encoder.

2.5.1 From Video Sequence to VLCs

Figure 2.8 is a visualization of the last stages in the H.26L video coder. In the H.26L core, motion estimation and transform coding take place. The transform coefficients, motion vector data and mode & sync data is then shipped to the modeling stage.

Recall from Section 1.1.4 that by modeling, we assign probabilities to different outcomes, or symbols, of the source. This is true for the entire underlying model, which permeates

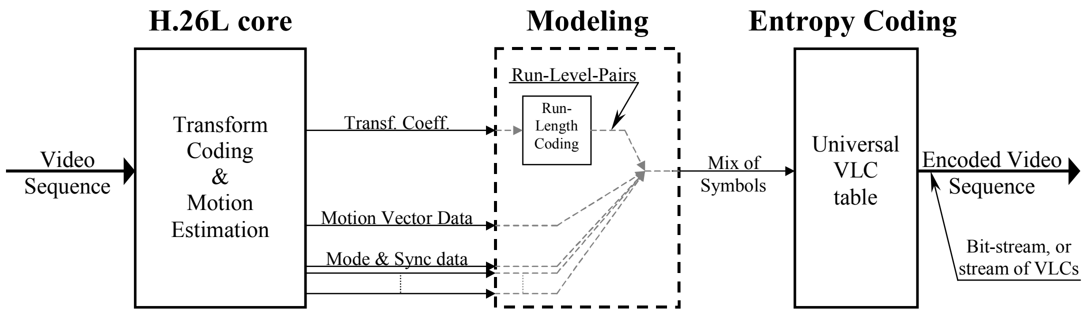

Figure 2.8 Final stages in the H.26L encoding of a video sequence.

the H.26L encoder as a whole. The modeling stage in Figure 2.8, however, represents the last steps of the full modeling process taking place in the H.26L encoder. Here it is not possible to affect the probability distributions of transform coefficients and motion vector data, etc., which are already assigned inherently by the H.26L core. It is still possible, however, to influence the efficiency of the encoding by adopting different models. This we will see in future sections.

One could argue that run-length coding (see Section 1.1.4) does not belong in the modeling stage, because it is actually a coding technique. On the other hand, run-length coding translates a small number of unique symbols with high occurrence frequencies into a large number of unique symbols with low occurrence frequencies. Thus, the run-length coding is manipulating, or modifying the symbol stream a great deal, and this is very much what modeling is about. Therefore, the modeling stage is certainly a suitable place for the run-length coding to reside.

In the current H.26L encoder all the data types are mixed and sent to the entropy coding stage where a universal VLC table (see Section 1.2.6) is used to code/represent all the symbols regardless of data type.

2.5.2 Definition of Entropy

Entropy is a measure of the information in a source. Intuitively, entropy can be seen as a measure of how uncertain we are of the outcome of the source. If there are N possible outcomes of the source, and each outcome has a probability of p_i, the entropy E is defined as

$$E = -\sum_{i=1}^{N} p_i \log_2 p_i \text{ bits.}$$

The base of the logarithm defines the unit by which the information is measured. Here logs are taken with base 2 giving the unit bits, which is the unit for a source coded with a binary alphabet, i.e. the two letters 0 and 1. From the definition of entropy we notice that E is a continuous function of p_i, and if each event is equally likely, E is a steadily increasing function of N. Furthermore, the entropy of a source is a function of the distribution of the symbol probabilities of the source. For a source with a fixed number of unique symbols N, the closer to a uniform distribution the distribution of the symbol probabilities is, the larger is the source's entropy. Consequently, the more skewed the distribution is the smaller is the source's entropy.

We can also calculate the entropy of a *symbol* in the source. If p_i is the probability that symbol i will occur in the source, the symbol entropy E_i is calculated as

$$E_i = -\log_2 p_i \text{ bits.}$$

This can be seen as a measure of how unlikely it is that symbol i will occur in the source. We say that more likely symbols, having greater probabilities, contain less information. Or in other words, the more surprising it is to encounter symbol i in the source, the more information the symbol carries and the larger is its entropy E_i.

When Huffman or VLC coding is employed each symbol i in the source is coded with an integral number of bits. However, the symbol entropy E_i is in most cases not an integral number and thus, it is very likely that some, or even many, of the symbols are coded using more bits then what is actually needed. This is not a problem for the arithmetic coding technique, since we are not bounded to use an integral number of bits for each symbol. In fact, the entropy E_i is equal to the number of bits needed to encode a symbol using **perfect** arithmetic coding.

2.5.3 Entropy with Respect to a Model

Any information source, whether it is a book, a binary file on a computer or a video sequence, has a *true* entropy. The true entropy of an information source is, however, extremely difficult to acquire. Efforts have been made to estimate the entropy of for example the English language. The procedure to establish such an estimate is to let test subjects read for example five pages in a book, and then guess a series of letters in the sixth page, one by one. Through this method the entropy of the English language has been found to lie in between 1.25 and 1.35 bits/letter (or bits/symbol).

It might be possible to make an estimate of the entropy of a certain video sequence in a similar manner. Test subjects could be shown say 50 frames of a video sequence, and then asked to make guesses about the contents of the 51st frame. This content, or information, is typically color and intensity of pixels, color and intensity of areas of pixels or simply motion in general. However, the method of using guessing becomes more abstract and difficult for video sequences compared to the English language. Nevertheless, every video sequence has a true entropy, which is the lower limit for how much it can be compressed without a loss of information and a decrease of image quality.

Instead of continue the daunting task of finding the true entropy we need to find other ways of utilizing the entropy measurement. The normal procedure is to design a model for the information source and calculate the entropy **with respect to the model**. Then, by improving the model, the entropy with respect to the model can be pushed closer to the true entropy.

When a video sequence is encoded, it undergoes a number of manipulations at different stages in the encoding process. Between each stage, the actual information in the source is represented in different ways. Every time the representation of the source changes, the entropy with respect to the representation changes. In some cases it is easy to calculate these entropies and compare them, which gives a powerful tool for tracking the exploitation of the redundancy in the source throughout the encoding.

When the word *entropy* is used in coming sections, it refers to the entropy with respect to the model, which is calculated right before the encoding process reaches the entropy coding stage. Figure 2.9 shows the flow of the H.26L encoding process just like Figure 2.8, but with the entropy coding stage removed. This is to point out the location of the source for which the entropy with respect to the model is calculated.

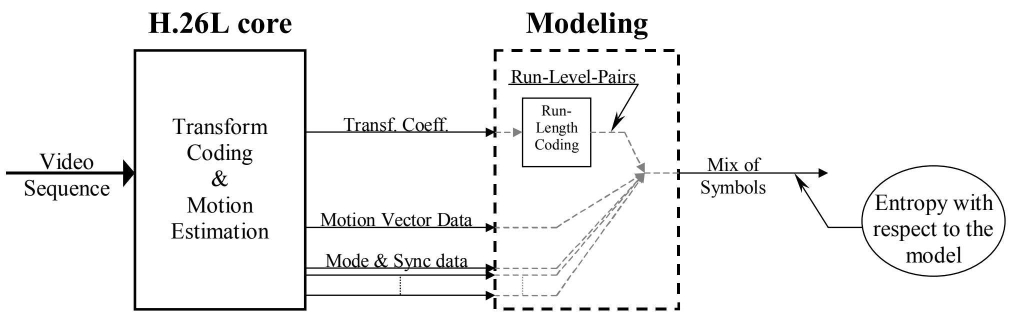

Figure 2.9 Location of source used for calculation of entropy with respect to the model.

2.5.4 Average Codeword Length (ACL)

The average codeword length is the metric used to measure the coding efficiency of a source that is entropy coded using a VLC table. The length of a codeword is the number of bits in the codeword. The average codeword length of a source with N different symbols, symbol probabilities $p_1, p_2, \ldots, p_N$ and codeword lengths $c_1, c_2, \ldots, c_N$ is calculated as

$$ACL = \sum_{i=1}^{N} p_i c_i \text{ bits.}$$

The lower the average codeword length the more efficient is the coding, and thus, the higher compression is achieved. The entropy of the source is the lower bound for the average codeword length for error free coding/decoding. Figure 2.10 shows that the source for calculating the average codeword length, naturally is the final H.26L output bit-stream.

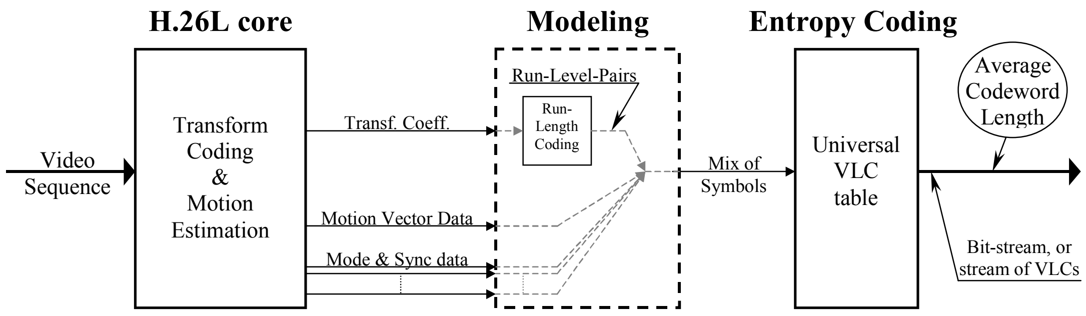

Figure 2.10 Location of source used for calculation of average codeword length.

2.5.5 Average Disjoint Entropy (ADE)

As stated earlier, the current H.26L version uses one universal VLC table to code all data types. It is common, however, to use multiple VLC tables where each table is optimized for a specific data type. This is done for example in H.263.

Before the source that is being processed by the H.26L video encoder enters the modeling stage, it can be viewed as *M* separate sub-sources where each sub-source corresponds to a specific data type or syntax element (see Figure 2.11). The entropy E_s can then be calculated for each sub-source. If p_s is the probability that a symbol belongs to sub-source *s*, a sum can be formed as

$$ADE = \sum_{s=1}^{M} p_s E_s \text{ bits.}$$

This sum will be called the average disjoint entropy (ADE). If c_s is the number of symbols in sub-source *s*, and c_{tot} is the total number of symbols in all sub-sources, p_s can be calculated according to

$$p_s = \frac{c_s}{c_{tot}}.$$

Figure 2.11 shows the location of the sub-sources used to calculate the average disjoint entropy. Instead of mixing all the symbols in the sub-sources and code/represent them using

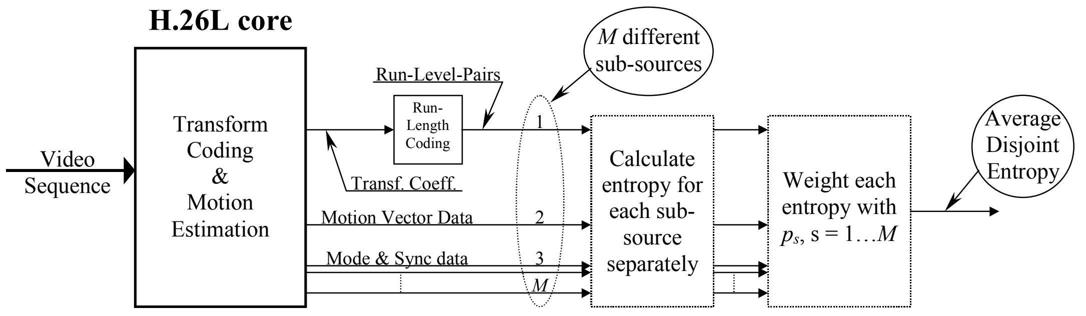

Figure 2.11 Location of source (or sub-sources) used for calculation of average disjoint entropy.

one universal VLC table, we could tailor different VLC tables for each sub-source. This is how it is done in H.263 and many other video coding standards. The result is a more efficient entropy coding, with the average disjoint entropy as the lower bound for the average codeword length. Recall from Section 1.2.6, however, that the design of the universal VLC table in H.26L is optimized for fast resynchronization in case of bit-errors. Another benefit of using only one regular VLC table is the decrease in complexity in both encoder and decoder.

2.5.6 No Double Scan

In the tests presented below, the double scan feature (see Section 1.2.6) has not been employed. The idea of double scan is to modify the model of the source, and thus the entropy with respect to the model, to correlate better with the one universal VLC table. The goal of this section, besides to study the efficiency of the universal VLC table, is to study the efficiency of the basic H.26L model and the entropy with respect to this model, **independently** of entropy coding technique. Clearly we have to examine the statistics of the source prior to an eventual usage of double scan. In the original H.26L codec software, double scan is adopted for INTRA-coded blocks with a quantization step below a certain threshold. When 300 frames are encoded and decoded in each run, as for the tests below, and only the first frame is INTRA-coded, the effects of double scan in the calculation of entropy, ACL and ADE are negligible. However, inhibition of double scan is of theoretical importance, and if, for instance, only INTRA-coding is considered, the effects of double scan have to be taken into account as well.

2.5.7 Entropy, ACL and ADE for H.26L and Foreman

Figure 2.12 shows the entropy for the QCIF format of Foreman encoded with H.26L. The ACL and ADE are included for comparison. As we can see in Figure 2.12, there is only a small difference between the entropy and the ACL, which means that the employed universal VLC table is well suited for coding all the different data types.

Observe that what is interesting about Figure 2.12 is not the increase in entropy with increasing bit-rate, but the difference between the three curves, entropy, ACL and ADE. The phenomenon that the entropy increases with increasing bit-rate is due to two things: a) the distribution of the symbol probabilities becomes more uniform as the bit-rate increases, and b) the number of unique symbols from the source increases with the bit-rate.

Figures 2.13a and b show exclusively the differences between entropy, ACL and ADE. These two curves are important because they give valuable hints about how much the bit-rate can be reduced through modifications of the entropy coding. However, remember that Figure 2.13a and b only show the results for the Foreman video sequence in the QCIF format, and thus, only tell us about possible bit-rate reductions for that particular video sequence. The purpose of these results is to give an idea of the efficiency of the original H.26L codec.

The entropy coding technique used in H.26L achieves remarkably good results for lower bit-rates, considering that only one VLC table is used. We see, for example, in the area around 100 kbits/sec in Figure 2.13b, that even if we tailored an **ideal** entropy coder to each

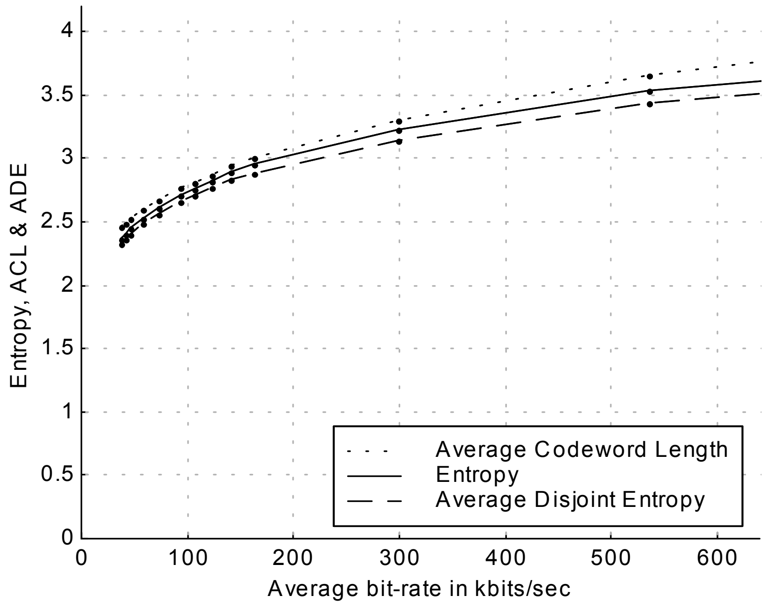

Figure 2.12 Entropy, ACL and ADE for H.26L (Foreman, QCIF)

data type for the Foreman video sequence, we would reduce the bit-rate only about four percent. Other sequences give similar results.

If the goal is to achieve further improvements for an arbitrary video sequence, and without modifying the H.26L core, this can be done in two ways. First of all, the modeling can be made more sophisticated, for example by using modeling contexts to detect correlation

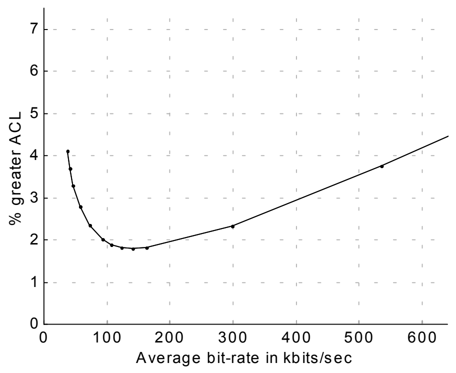

Figure 2.13a Difference between entropy and ACL.

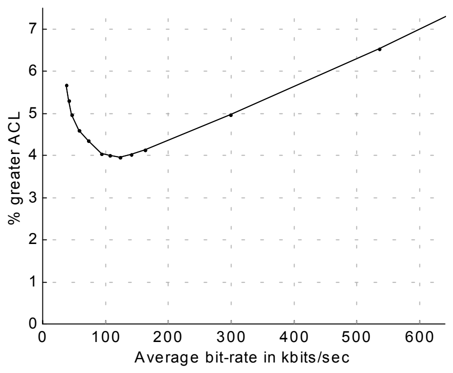

Figure 2.13b Difference between ADE and ACL.

between different data better, and thereby exploit redundancy more efficiently. Second, we can let the entropy coding, or the model, adapt to local statistics (spatially or temporally local). An adaptive model alters the representation of the data during encoding (a primitive example of this is the double scan feature). An adaptive entropy coder detects local changes in the distribution of symbols, and optimizes the representation of the symbols thereafter. Adaptive arithmetic coding is a good example of adaptive entropy coding, however, other techniques exist as well and will be discussed later.

2.5.8 Difference Between Entropy and ACL for Different Data Types

Figures 2.15a-f show, for different data types, how much greater in percent the ACL is compared to the entropy for Foreman in the QCIF format. This is equivalent to the coding efficiency of the data types, where lower values mean more efficient coding. Recall that the intra prediction mode data type is used only in INTRA-pictures and that motion vectors only appear in INTER-pictures**. Notice also the different scales on the Y-axes**. Figure 2.14 shows a typical distribution of bits over different data types for 300 frames of a video sequence

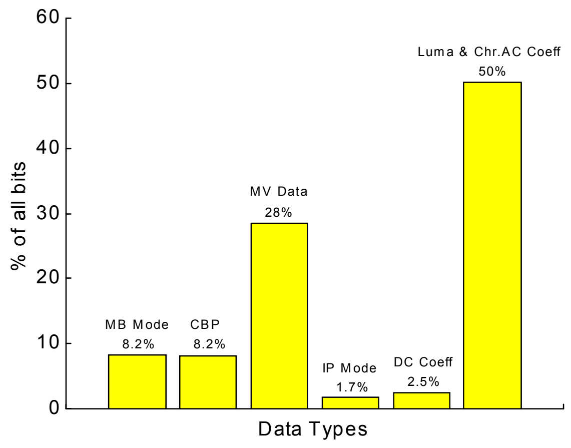

Figure 2.14 Typical distribution of bits over different data types.

encoded with the H.26L encoder resulting in a coded video sequence with moderately good image quality. Use the distribution in Figure 2.14 to weight the importance of the data types in Figures 2.15a-f.

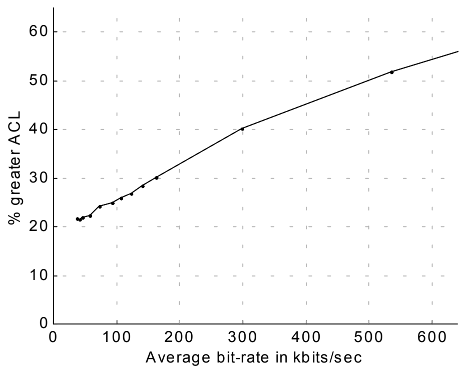

Figure 2.15a MB type

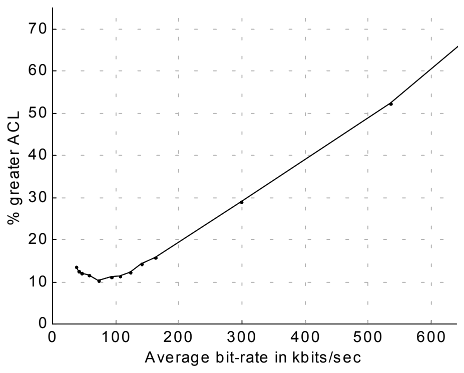

Figure 2.15b CBP

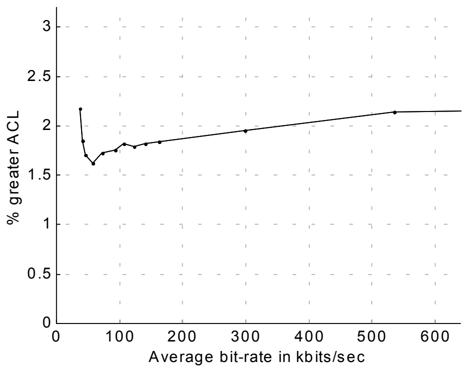

Figure 2.15c Motion vector data

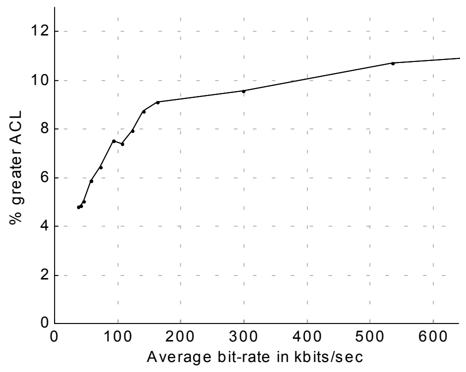

Figure 2.15d Intra prediction mode

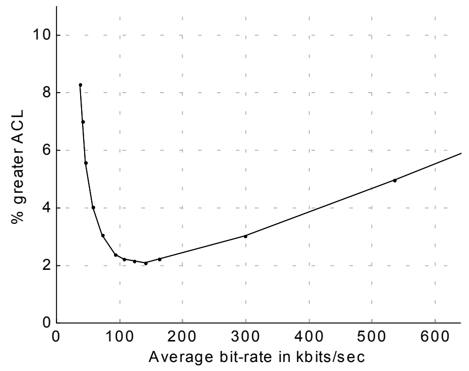

Figure 2.15e Luma & chroma AC coefficients

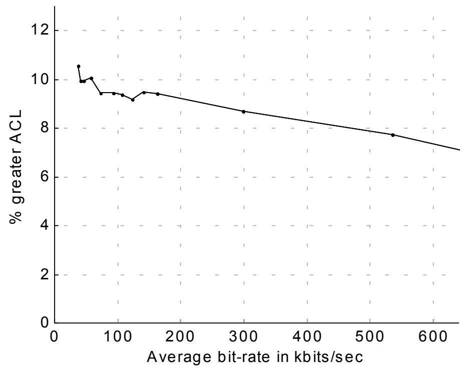

Figure 2.15f Chroma DC coefficients

Data types such as MB Type and CBP are, on average, coded inefficiently. Transform coefficients and intra prediction symbols are coded efficiently, and motion vectors are coded very efficiently. It is clear that H.26L is optimized for compressing video sequences to a bit-rate of around 100 kbits/sec. This is especially prominent in Figure 2.15b showing the coding efficiency for the CBP data type, which is fairly good around 100 kbits/sec and then rapidly decreases with increasing bit-rate.

2.6 Evaluation Conclusions

From the video codec evaluation presented above can be drawn four main conclusions corresponding to the four different test areas. These conclusions are addressed in Sections 2.6.1-4 below.

2.6.1 H.26L versus H.263

We learned in Section 2.1 that the H.26L codec outperforms the H.263 codec. However, the gain in SNR for H.26L compared to H.263 is less for lower bit-rates, which are often of more interest. Furthermore, Version 1 of the H.263 video codec has been developed and improved, and later versions achieve better compression results in general at all bit-rates.

It is clear that more improvements have to be made to the H.26L video codec in order to reach ITU-T's goal of 50% greater bit-rate savings from H.263. Recently an entirely new proposal was presented for the H.26L standard [18]. This new video codec is built on a more sophisticated motion model that captures affine motion (see Section 1.3.3); the motion prediction algorithms are also considerably more advanced. Remarkably good results are achieved by the new proposal, almost in line with ITU-T's requirements. As far as entropy coding goes, the new video codec employs several individually optimized VLC tables, just like H.263.

2.6.2 Arithmetic Coding versus VLCs for H.263

Section 2.2 showed that for low bit-rates up to 5% better compression can be achieved for a typical video sequence by employing arithmetic coding for H.263. The bit-rate reductions are smaller for higher bit-rates. A bit-rate reduction of at most 5% may seem insignificant. However, the arithmetic coding technique in H.263 is simple and adds little complexity to the codec. If bit-rate savings of 3-5% can be made through rather simple means, that is of importance.

The results in Figure 2.5 and Figure 2.13a show that for the Foreman video sequence the H.263 arithmetic coding technique would be less efficient if used with H.26L, at least for lower bit-rates. This is because the universal VLC table in H.26L already codes the Foreman video sequence close to its entropy (see Figure 2.13a).

2.6.3 Bit Distribution for Different Data Types

A video codec like H.263 uses the vast majority of available bits to represent the transform coefficients. When this is the case, and the object is to develop an improved entropy coding technique, it may be a good idea to tailor the coding technique to the transform coefficients. Even if that coding technique is not optimal for other data types it may very well be the overall optimal solution to a system where the same entropy coding technique is employed over all data types. We can see in Figure 2.6, however, that for H.26L the distribution of bits over different data types is rather even, and concern has to be taken to all of the main data types, such as transform coefficients, motion vector data and CBP. Otherwise it is possible that the improved entropy coding technique will give poor results, especially for lower bit-rates where the distribution is more even.

2.6.4 Entropy Study of the H.26L output

The entropy study of H.26L showed a number of interesting things. First of all, we found that the universal VLC table currently used in H.26L is not as inefficient as it may seem. Thus, improving only the entropy coding stage and not the modeling, for example by employing arithmetic coding, is not of interest (compare Figure 2.13a). Second, we learned that with different modeling **and** entropy coding it should be possible to improve the efficiency to a degree where it becomes interesting (compare Figure 2.13b). For certain video sequences and bit-rates it might be possible to reduce the bit-rate over 5%, which is not negligible.

The next two sections will describe the adaptive arithmetic coding technique and different modeling techniques and evaluate those techniques in combination with the H.26L video codec.

3. The Adaptive Arithmetic Codec

This section begins with a discussion about adaptive methods in general, and then follows a description of the adaptive arithmetic codec that has been integrated with the H.26L video codec. It is important to have knowledge about the functionality of the adaptive arithmetic codec (Section 3.2) in order to understand its drawbacks and benefits. We also need to understand the main parameters of the adaptive arithmetic codec (Section 3.3) to be able to design efficient models.

3.1 Adaptive Methods

Adaptive methods are a powerful tool when exploiting local statistics. Adaptation is a very wide spread and well-known technique that can be used for many purposes in many different areas. Which statistics we choose to adapt to is dependent on what we wish to achieve with the adaptation. We may, for example, want to change the coding or representation of a video sequence by adapting to the characteristics of a transmission channel to create a more robust transmission. Below is an overview of three examples of adaptive methods incorporated into video codecs in order to reach a lower bit-rate for a given image quality.

3.1.1 Dynamic Symbol Reordering in H.26L

In October 1999, an enhanced entropy coding method for H.26L based on dynamic symbol reordering (DSR) was presented [15]. The method examines the symbol statistics for previous frames (global DSR) and macroblocks (local DSR) and reorders the symbols on the fly so that the most probable symbols are coded using the shortest codewords. The same universal VLC table is still adopted. The document reports a bit-rate reduction of about 8% on average using this adaptive algorithm. These bit-rate reductions are more or less independent on bit-rate.

The DSR based method typically adds complexity to both encoder and decoder. A more important drawback, however, is that the method creates dependencies between frames, since the global DSR collects statistics from several previous frames. The results of this is that if a bit-error occurs in the bit-stream, which results in an incorrectly decoded symbol, **all** the subsequent frames will be affected. When the error is actually detected it might be very difficult to track the location where the error occurred and thus difficult to know how many frames need to be retransmitted. In other words, it is not possible to create an error resilient bit-stream using the DSR based method presented in [15]. However, if only the local DSR is employed and not the global DSR, there would be no dependencies between frames and thus, the possibility to create an error resilient bit-stream would be increased significantly.

3.1.2 Adaptive Arithmetic Coding in Wavelet-Transform Video Coder

As mentioned in Section 1.3.4, adaptive methods combined with arithmetic coding makes an elegant and useful system. The system is complex, but not overly complex. Adaptive arithmetic coding is frequently employed in, for example, image compression [7], but has been used sparsely in video coding algorithms. There exist, however, examples of adaptive arithmetic coding in combination with video compression. In [6] is presented an algorithm for very low bit-rate video coding using wavelet-transforms. Here wavelet-transform coefficients and motion vectors are coded using adaptive arithmetic coding. The gains in compression efficiency due to the adaptive arithmetic coding are difficult to measure and not reported exclusively. However, it is claimed that the video coding algorithm is superior to the H.263 recommendation. Adaptive arithmetic coding was integrated into the core algorithms from the very beginning of the development of the video coder, which makes it able to fully exploit the advantages of adaptive arithmetic coding. It is interesting to note, for example, that it is not necessary to predict motion vectors because the local statistics of motion vector components are well exploited by the adaptive arithmetic coder.

3.1.3 Multiple VLC Tables for Transform Coefficients in H.263

In contrast to H.26L, H.263 uses multiple VLC tables. Each VLC table corresponds to a specific data type or syntax element. In [9] a method for H.263 is presented where up to 63 VLC tables are used to encode transform coefficients. In H.263, macroblocks are partitioned into four 8x8 sub-blocks on which the DCT transform is performed. The transformed two dimensional sub-block is then reordered into a one dimensional vector by the means of a scan pattern (typically a zig-zag scan depicted for a 4x4 block in Figure 1.6a). The VLC table used to encode a transform coefficient (and its preceding run of zeros) is decided by the position of the last preceding non-zero coefficient in the vector. For instance, let us say that coefficient number 55 in the vector is non-zero, followed by three zero coefficients and then coefficient number 59 is non-zero. To encode coefficient number 59 we use the one of the 63 VLC tables that is associated with coefficient 55. Which entry in this VLC table is used, is then decided by the three zero-coefficients and the level of coefficient number 59 (regular run-length coding). In this way, statistical redundancies between transform coefficients are exploited. Bit-rates savings from 5% for very low bit-rates up to 18% for higher bit-rates are reported.

Now, one could discuss whether or not the method presented in [9] should be called an adaptive method. However, it is an interesting and innovative method, which does not need any additional information to be sent between encoder and decoder, and it gives significant bit-rate savings.

3.2 Adaptive Arithmetic Codec Functionality

The adaptive arithmetic codec replaces the universal VLC table used in H.26L. Figure 3.1 shows the H.26L encoding process just like Figure 2.8, but with adaptive arithmetic coding in the entropy coding stage. The modeling will be dealt with in Section 4, and only the adaptive arithmetic codec by it self is described here.

The choice was made not to implement an adaptive arithmetic codec from scratch, but to base it on the adaptive arithmetic codec described in [22]. Along with [22] comes an implementation in the C language, which is available on the Internet (refer to [22]).
The codec described in [22] was originally made for text compression. However, it is a

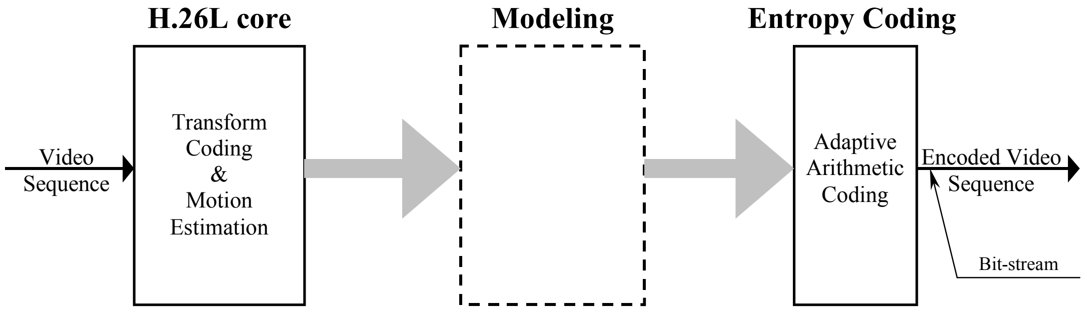

Figure 3.1 Adaptive arithmetic coding instead of universal VLC table.

generic codec that encodes and decodes *symbols* in general. These symbols can, instead of representing letters or characters, represent for example transform coefficients and motion vectors.

The main goal of this section is to describe the adaptive arithmetic codec that has been integrated with the H.26L video codec, with emphasis on the adaptation process. The arithmetic coding is already described in Section 1.1.4.

The adaptation process alone is identical in the encoder and the decoder. Therefore, whenever the adaptation functionality of the encoder is mentioned below, the same applies to the decoder as well.

3.2.1 Model and Symbols

The core of the adaptive arithmetic codec is independent of the model, however, an interface between the model and the adaptive arithmetic codec has to be designed. In order to do this it is necessary to have knowledge about the symbols that will be transferred from the modeling stage to the adaptive arithmetic coding stage (Figure 3.1). Models are, however, further discussed in Section 4, and in this section it is assumed that the model is already known.

3.2.2 Contexts and Adaptation

The adaptive arithmetic codec uses contexts to organize data. A context consists of a set of predefined symbols and one frequency counter per symbol. The frequency counter is used to count the occurrence of a symbol during encoding. Symbols are, when the context is created, manually installed one by one. By default the frequency count is initially set to 1 for each symbol. When a context has been created the encoding can be started, and each time a symbol is encoded the corresponding frequency count is updated. The frequency counters are by default incremented by 1 when updated. Through this updating process each context keeps its own probability distribution, which adapts to the symbol statistics during encoding. The probability distribution is then used for the arithmetic coding of the symbols (refer to Section 1.1.4).

The frequency counts of the symbols are maintained cumulatively in a tree-based structure. As mentioned in Section 1.1.4 it is undesirable to allow arbitrarily small intervals, or symbol probabilities, in the arithmetic codec and thus, the precision B is in the current version of the adaptive arithmetic codec set to the maximum value of 32 bits (assuming 32 bit integers). Due to the functionality of the arithmetic codec's encoding/decoding algorithms (refer to Section 1.1.4), the precision B needs to be four times smaller than the smallest possible interval, or symbol probability, in order for each symbol to be correctly decoded. Therefore, the total frequency for each context, i.e. the sum of all frequency counts in a context, can be up to B-2 bits = 30 bits. This means that the total frequency may reach a value as large as 2^{30} = 1,073,741,824. In many cases this is more than enough. Should, however, the total frequency reach its maximum value, all the frequency counts will be halved (uneven frequency counts are rounded up). For the three models presented in Section 4, 30 bits are certainly enough for no frequency halving to occur.

The number of contexts that should be used in the adaptive arithmetic codec is determined by the model. We need to use one particular context for each set of symbols that we wish to adapt a unique symbol distribution for. Contexts could for example correspond to different data types or syntax elements, or a number of adjacent macroblocks with similar properties. Refer to Section 4.1 for a further discussion about contexts.

3.2.3 Escape Symbols

Since contexts can be created to correspond to symbols in a certain data type or to symbols having certain characteristics or fulfilling certain properties, the number of (unique) symbols in a context might not be known or might be very large. For a context with a finite, but very large amount of symbols, it might be difficult to gain knowledge about all the possible symbols. Still, when a context is created it needs to have all its symbols predefined. That is, each unique symbol needs to be installed in the context prior to the start of the encoding. This is because the current version does not allow symbols to be installed on the fly in order to keep down the complexity of the adaptive arithmetic codec.

Now, the question is how to deal with the symbols that according to the definition of the context belong to the context, but are not installed in the context. This problem is solved by installing an *escape symbol* in the context. When the encoder encounters a symbol that is not installed in the context it transmits the escape symbol, telling the decoder that the following symbol is not installed in the context and will be transmitted differently. Next, the encoder sends the symbol for example by using normal binary representation (the encoder and decoder have to agree upon the number of bits used for this representation).

3.2.4 Purging Contexts

At any time during encoding or decoding a context can be purged, which means the context is re-initialized back to the state it was in immediately after it was created. This is important for error resilience. Since it is crucial for correct adaptation that all the symbols are decoded correctly, a bit-error in the bit-stream resulting in an incorrectly decoded symbol means that the decoding has to be interrupted and restarted where the adaptation begun. Thus, never purging contexts means that the entire video sequence have to be retransmitted in case of an error. However, in order to utilize the benefits of adaptation the purging should not be done too frequently.

3.2.5 Stopping and Restarting the Adaptive Arithmetic Codec

As mentioned in Section 3.2.2, the precision *B* is in the current version of the adaptive arithmetic codec set to 32 bits. We learned in Section 1.1.4 that the decoder keeps a window of *B* bits (here 32 bits), which means that whenever the decoding is stopped there will be approximately 32 bits in the window that are discarded since they are not necessary for the decoding. These redundant bits have to be transmitted by the encoder. The consequence of this is that when the adaptive arithmetic encoder/decoder is stopped and restarted, for example for sending a synchronization codeword or an escape symbol using normal binary representation, it costs approximately 32 bits each time.

There is a way to make the adaptive arithmetic codec less wasteful of bits when stopped and restarted. Instead of letting the encoder transmit the padding bits, we let the decoder read *B* bits past valid coding output (possibly past end of file). Then, when the decoding is stopped, the unnecessary bits read by the decoder are put back to the input bit-stream. However, this solution is not entirely trivial, and for simplicity the current version of the adaptive arithmetic codec uses the extra 32 padding bits each time it is stopped and restarted.

3.2.6 Manual Modification of the Symbol Distribution in a Context

To gain control over the adaptation process, a new function called *update_symb_freq* was added to the adaptive arithmetic codec, which affects the symbol distribution in a context. The function updates the frequency count of a symbol without affecting the output bit-stream (i.e. without sending any bits to the decoder). This way, the symbol distribution in a context can be set regardless of which symbols have been or are being encoded. This is useful for example if the contexts are frequently purged and we do not want to re-initialize the symbol's frequency counts to 1. Re-initialization is further discussed in the following section.

3.3 Adaptive Arithmetic Codec Parameters

When models are created for the adaptive arithmetic codec, there are four main parameters, or initializations, that we need to consider. These parameters have a direct impact on coding efficiency. Although, the impact varies with type of video sequence, type of model and for different parameters.

3.3.1 Context Sizes

There is a trade-off between context size and coding efficiency. Since all symbols in a context share the probability space from 0 to 1 (Table 1.1b), installing unnecessary many symbols in a context results in small intervals which require more bits to be coded. However, the degradation in coding efficiency due to large contexts is usually quite small since the intervals are altered during adaptation.

Too small contexts result in a more frequent use of escape symbols, which makes the coding less efficient.

Experiments using different context sizes show that the best results seem to be achieved when contexts include the vast majority of possible symbols and that only the most uncommon symbols are coded differently using escape symbols.

3.3.2 Initial Symbol Distribution

Each time a context is purged the frequency counts for all the symbols in the context are by default set to 1. That is, the probability distribution of all the symbols is uniform, meaning that all symbols are equally likely to occur. In general this is not the case, and some of the symbols are more likely than other symbols. The adaptation routine will eventually build up a new symbol distribution, but before this is done the coding of the symbols will be inefficient, especially if the true symbol distribution is very skewed.

In the current version of the adaptive arithmetic codec, contexts are manually initialized using the *update_symb_freq* function. Let us say, for instance, that we want to initialize a context containing 10 symbols with the symbol distribution in Table 3.1. The corresponding frequency counts, appearing in the right most column in Table 3.1, are then set by the *update_symb_freq* function. The result is a context with an **exact** copy of the symbol distribution as it would be if the symbols had been encoded normally.

It would be preferable to initialize a newly purged context with a symbol distribution that is typical for that particular context. Unfortunately this implies several problems. How do we establish a typical symbol distribution? How dependent on quantization step is a context's symbol distribution? How dependent on the video sequence (or the amount of motion in the video sequence) and the format of the video sequence is a context's symbol distribution? These questions may demand, for each context, a substantial amount of tests and experiments to be answered.

Table 3.1 Symbols, probabilities and corresponding frequency counts.

Symbol	Probability	Frequency count
a	0.6	600
b	0.2	200
c	0.1	100
d	0.06	60
e	0.02	20
f	0.01	10
g	0.005	5
h	0.003	3
i	0.001	1
j	0.001	1

3.3.3 Re-initialization Frequency

As mentioned earlier, a bit-stream can never be made error resilient if contexts are not purged and re-initialized. The less frequently a context is purged and re-initialized the more data has to be retransmitted or discarded in case of a bit-error. Discarding data is necessary in a system with delay constraints not allowing costly retransmissions. For instance, if the video sequence is to be used in a conversational application for real-time communication over an error prone channel, it is exceptionally important to frequently purge and re-initialize contexts, so that it is possibly to achieve an error resilient bit-stream.

Error resilience is, however, not the only reason why purging and re-initialization is important. One of the goals of the adaptive arithmetic coding is to constantly enhance the coding to be optimal for the temporally local statistics. If the adaptation is never reset, the local statistics will become negligible compared to the overall global statistics in the video sequence, and the correlation between temporally local symbols will not be exploited. On the other hand, if contexts are purged and re-initialized too frequently, the gain in compression efficiency due to adaptation will be lost. If error resilience was not a concern, the problem could be solved by not resetting the adaptation but to scale down all the frequency counts so that local statistics would become more prevalent but the symbol distribution built up through adaptation still would contribute.

The variations in local statistics are highly dependent on the video sequence. A certain re-initialization frequency might be good for one particular video sequence and unsatisfactory for another. In the current version of the adaptive arithmetic codec, the re-initialization frequency can not be changed during encoding/decoding. For simplicity the re-initialization frequency is the same for all the models presented in Section 4, and contexts are purged and re-initialized between frames. In this way we make use of as much statistics as possible without creating dependencies between frames.

3.3.4 Adaptation Rate

Initializing contexts with a typical symbol distribution before the encoding starts imposes a problem on the adaptation process. The frequency counts in Table 3.1 represents 1000 symbols. That is, if the symbol distribution in Table 3.1 were to be built up through encoding (and not manually using *update_symb_freq*), 1000 symbols would have to pass through the encoder. The consequence of this is that when the encoding and thus the normal adaptation starts, not until 1000 symbols have been encoded will the part of the total symbol distribution which is due to adaptation, affect the arithmetic coding of the symbols as much as the initial symbol distribution. Whether 1000 symbols is a significant amount or not is of course highly dependent on how many symbols are encoded before the context is purged and re-initialized. If for example 10000 symbols are encoded before re-initialization the symbol distribution built up from adaptation would, right before the next re-initialization, have 10 times more influence on the arithmetic coding than the initial symbol distribution. If only 100 symbols are encoded before re-initialization, we have the opposite situation, and the influence, or gain due to adaptation would, on average, be almost negligible. We need a method to control the contexts so that there is a certain gain from adaptation in the same time as we achieve efficient compression in the beginning of the encoding. Three solutions are discussed below:

1. One way to gain full control over how many symbols have been encoded when a context is purged and re-initialized, is to let the re-initialization frequency be a function of the number of encoded symbols. To achieve this, we could, in a variable *num*, store the amount of symbols that the initial symbol distribution corresponds to, and re-initialize the context when we have encoded some multiple of *num*, for example when 20×*num* symbols have been encoded. There are, however, a few major drawbacks of this method. It gives us no control over when contexts are purged and re-initialized since we do not know during encoding how many symbols the video sequence will produce. Furthermore, the re-initialization of contexts would be asynchronous, which is non-preferable in an error resilience point of view, and for some contexts the re-initialization frequency might not be high enough to prevent dependency between frames.

2. Another solution to the problem is to keep two symbol distributions for each context. One that is a typical initial symbol distribution and one that starts out "blank" and is built up through adaptation during encoding. The symbol distributions (frequency counts) are then mixed (added) but weighted differently according to which symbol distribution we want to affect the arithmetic coding the most. We weight the initial symbol distribution heavier in the beginning of the encoding and slowly move the weight over to the adapting symbol distribution. The drawbacks of this method are that we do not know what rate to use for the change and the even more advanced adaptation scheme adds complexity to both the encoder and the decoder.

3. In the current version of the adaptive arithmetic codec, the problem regarding adaptation and initial symbol distribution is solved by weighting the importance of symbols being encoded heavier than the symbols corresponding to the initial symbol distribution. When a symbol is encoded it is counted more than once, i.e. a symbol's frequency count is not incremented by 1 as is default, but by a user defined constant greater than 1. This constant is stored in a variable called ADAPT_RATE. In this way we only need to keep one symbol distribution in the context and can control how fast the symbol distribution adapts to local statistics. Unfortunately it is difficult to know the optimal setting of ADAPT_RATE.

4. Models and Results

The design and the results for three different models are presented below. The models were designed in the same order as presented, and each model gives valuable information and hints about what should be modified and improved in the next model.

Models have been designed with the assumption that INTER-pictures are much more common than INTRA-pictures, and the majority of the work has been focused on improving the INTER-coding. When the models are evaluated, 300 frames of the video sequences are used. For most bit-rates this means that the effect of the INTRA-coding is negligible since only one out of 300 frames is INTRA-coded. Nevertheless, adaptive arithmetic coding is used for INTRA-pictures as well and in the few tests made to evaluate the INTRA-coding solely, the bit-rate was reduced somewhat and never increased. However, due to the great emphasis on INTER-coding, the INTRA-coding parts of the models are not presented.

Recall that purging and re-initialization of every context occurs once per frame for all models, right before the encoding/decoding of the frame (Section 3.3.3).

4.1 Designing Models

Before the models are presented it is good to have some knowledge about general model design (design of models was also briefly addressed in Section 1.3.4). An important goal for a model is to exploit correlation between data. When designing a model for video coding, our major concern will be to find/predict spatial correlation, i.e. correlation within a frame. We could for example look for correlation between adjacent pixels, transform coefficients, motion vectors, macroblocks or sub-blocks, etc. Assuming that adjacent elements are more likely to have similar characteristics than elements spatially far away from each other.

Let us use the macroblock case as an example. Figure 4.1 shows a group of macroblocks somewhere in a frame. Since frames are encoded/decoded from left to right and from top to

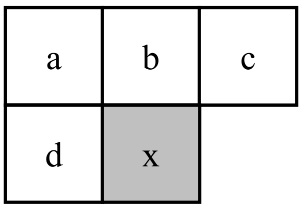

Figure 4.1 Macroblock *x* and adjacent macroblocks *a*, *b*, *c* and *d*.

bottom, macroblocks *a*, *b*, *c* and *d* will always be available in the encoder and decoder as macroblock *x* is being encoded/decoded (except when *x* correspond to certain macroblocks on picture edges). Thus, macroblocks *a*, *b*, *c* and *d* can be used in what we call a modeling context, or just context, to predict aspects about macroblock *x*, such as number of motion vectors, skipped or non-skipped, etc. It is for example natural to believe that if three or more of macroblocks *a*, *b*, *c* and *d* are skipped, it is more likely that macroblock *x* is also skipped, compared to if only one or none of macroblocks *a*, *b*, *c* and *d* is skipped. A context could obviously consist of more or less macroblocks than in the example above. For instance only macroblock *b* and *d* could form a context.

In [20], [25] and [27] methods are presented that use contexts in a similar way as described above. In [27] for example, previously encoded/decoded transform coefficients form different contexts.

Whether or not a context will efficiently exploit redundancies is directly dependent on how much correlation there is within the context. Unfortunately, this correlation differs with aspects like type of data, type of video sequence, type of video format, quantization step, how much is already exploited in preceding coding steps. As a first step, Model 1 presented below is designed using simple contexts which correspond directly to the data types in H.26L. This results in statistics from every preceding macroblock within a picture being used, instead of for example macroblock *a*, *b*, *c* and *d* in Figure 4.1.

4.2 Model 1

Figure 4.2 is a schematic figure of the modeling process for Model 1. Here we see that instead of mixing all the data types as the original H.26L model (Figure 2.8), each data type is coded separately. Important is that there is no run-length coding, which is addressed in the next section.

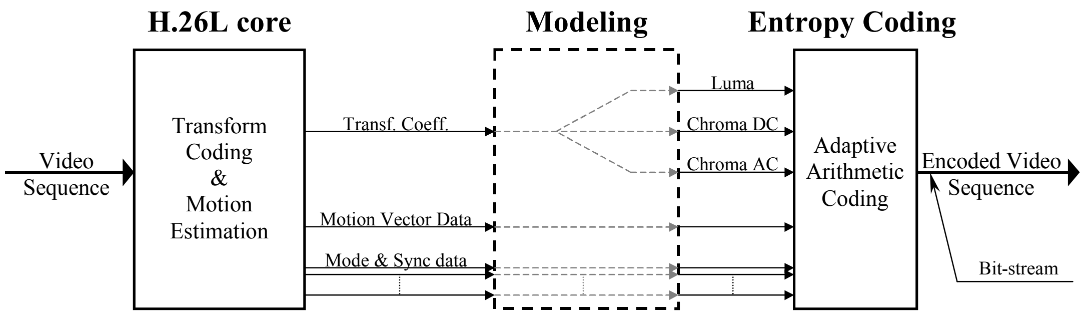

Figure 4.2 Schematic figure of the modeling process for Model 1.

4.2.1 No Run-Length Coding

According to Section 2.5.1, run-length coding aims to translate a small number of unique symbols with high occurrence frequencies into a large number of unique symbols with low occurrence frequencies. When using adaptive arithmetic coding we want to keep down the sizes of the contexts and divide the data into as small elements as possible. Small elements are beneficial because they result in a more efficient adaptation, provided that each single element contributes to the adaptation. In other words, a small number of unique symbols with high occurrence frequencies is in general preferable when using adaptive arithmetic coding. Because of this the run-length coding is not employed in Model 1 (see Figure 4.2).

4.2.2 Model 1 Contexts

Model 1 uses five contexts for INTER-pictures, which are listed in Table 4.1. The second column of Table 4.1 shows the context sizes, i.e. the number of (unique) symbols in each context. The third column shows how many symbols correspond to the initial symbol distributions in the contexts. We will refer to this as number of initial symbols (NOIS). According to the discussion in Section 3.3.4, we need to have knowledge about NOIS, since it has quite an impact on the adaptation process. Finally, the right most column gives a brief description of each context.

Table 4.1 Contexts for Model 1.

Context name	Context size	NOIS	Context description
MB_MODE_CONTEXT	8	99	Used to code MB_Type only
MVD_CONTEXT	60	373	Codes all the motion vector data
LUMA_CONTEXT_INTER	50	15165	Handles luma transf. coeff.
DC_CONTEXT_INTER	40	544	Handles chroma DC transf. coeff.
AC_CONTEXT_INTER	30	8161	Handles chroma AC transf. coeff.

Recall that the CBP data type described in Section 2.3.8 contains information about which sub-blocks and which coefficient types (luma, chroma DC, chroma AC) are to be coded for each macroblock. In model 1 this information is obsolete because all the transform coefficients, of all the three coefficient types, are transmitted for each non-skipped macroblock. Ignoring the extra cost associated with encoding and sending all the transform coefficients, approximately 7-10% is saved in bit-rate by not transmitting the CBP codewords (see Figure 2.6).

The five contexts in Table 4.1 simply correspond to the five most common data types. All of the possible MB_Type symbols are included in the MB_MODE_CONTEXT (it is possible to send INTRA-coded macroblocks in INTER-pictures, however, this feature has been turned off in both the original and the AAC-based codec). Hence, the size of MB_MODE_CONTEXT is 8 (compare Section 2.3.3). Not all the symbols are known by forehand for the MVD_CONTEXT and the three coefficient contexts and thus, escape symbols have to be used. Experiments were made to find optimal context sizes. Lower bit-rates were slightly prioritized resulting in small contexts and a considerable amount of escape symbols used for very high bit-rates.

4.2.3 Initial Symbol Distribution for Model 1

Because of the large amount of symbols being encoded in each frame, and hence, the large amount of statistics available between purging and re-initialization of the contexts, the coding efficiency was assumed to be less dependent on the initial symbol distributions. However, not to use a typical initial symbol distribution at all, i.e. to initialize all contexts with the frequency count 1 for all symbols, would be an unwise choice. We know for example that a zero transform coefficient is much more likely than a non-zero transform coefficient, resulting in a highly skewed symbol distribution. The initial symbol distribution used here was partly designed by hand and partly taken from an INTER-picture at a position with a fair amount of motion in a compressed video sequence with a moderate image quality. Furthermore, not all the 60, 50, 40 and 30 symbols in the contexts were included in the initial symbol distribution. Only the most likely symbols were included, the rest were left with a frequency count of 1.

4.2.4 ADAPT_RATE for Model 1

Without further investigation, the ADAPT_RATE was set to 5, meaning that symbols are counted once, plus five extra times, when encoded. That is, the frequency counts are incremented by 6 instead of the default value of 1. An adaptation rate of 5 might seem small compared to the large amount of symbols corresponding to the initial symbol distributions for the transform coefficients (for example 15165 for luma transform coefficients). We need to keep in mind, however, that run-length coding is not employed and the number of luma, chroma DC and chroma AC transform coefficients that are sent for each frame can be as high as 25344, 792 and 11880 respectively, for the QCIF format, provided that no macroblocks are skipped. However, for lower bit-rates and for video sequences with little motion, the corresponding figures are smaller, since several macroblocks are skipped.

At the end of the encoding of a frame, the symbol distribution built up from adaptation is weighted about 5-10 times heavier (for luma transform coefficients) than the initial symbol distribution, depending on how many macroblocks are skipped.

4.2.5 Bit-Rate Reduction for Model 1

Figure 4.3 and 4.4 show the bit-rate reductions for the Foreman and Mobile video sequence respectively. The Mobile video sequence introduced here is referred to as a high bit-rate video sequence. This is because it requires a significant amount of bits to code the Mobile video sequence compared to other video sequences. This is due to its high amount of motion.

It is clear from Figure 4.3 that there is something fundamentally wrong about Model 1. The bit-rate is not decreased, but rather increased, especially for low bit-rates. It is interesting however, how much better suited Model 1 is for the Mobile video sequence according to Figure 4.4. The model is poor for low and intermediate bit-rates, but for high and very high bit-rates the model reduces the bit-rate considerably, even above 15%.

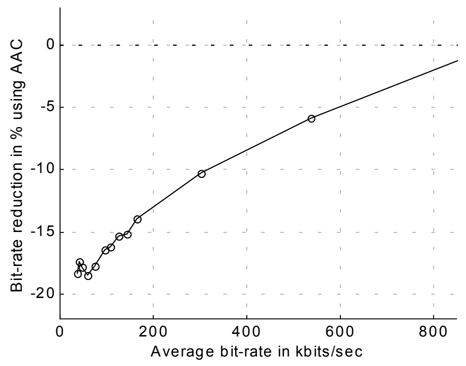

Figure 4.3 Bit-rate reduction for Foreman (QCIF)

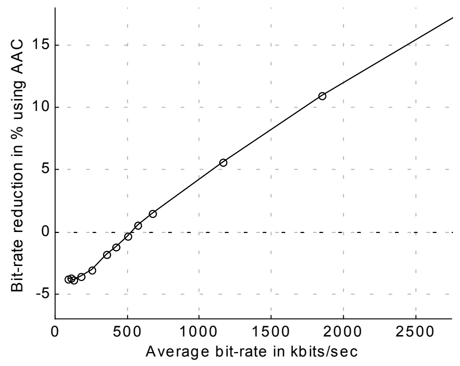

Figure 4.4 Bit-rate reduction for Mobile (QCIF)

4.2.6 Conclusions

The main difference between Model 1 and the original H.26L model is that Model 1 encodes and transmits all the transform coefficients, one by one, for non-skipped macroblocks. This means that significantly more symbols are sent from encoder to decoder. Furthermore, we see in Figure 4.3 and 4.4 that Model 1, on average, increases the bit-rate. It is natural to believe that there is a correlation between the handling of transform coefficients and the negative results. In Model 2, an effort was made to alleviate this problem.

4.3 Model 2

Figure 4.5 is a schematic figure of the modeling process for Model 2. As for Model 1, run-length coding is not employed. A new data type is used to control transform coefficients (in Figure 4.5 called TC Mode), this data type is described in the next section.

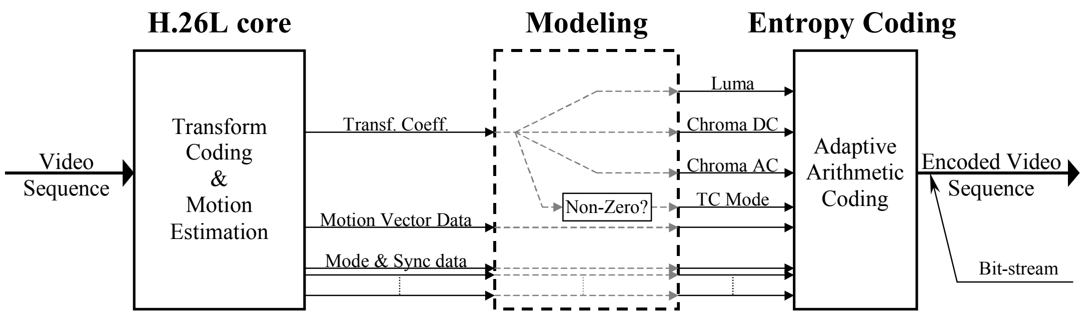

Figure 4.5 Schematic figure of the modeling process for Model 2.

4.3.1 Model 2 Contexts

The five contexts for the MB_Type data type, motion vector data and transform coefficients are for Model 2 identical to the contexts in Model 1. In Model 2 there are, however, three new contexts used by the encoder to inform the decoder about which transform coefficients are being encoded and transmitted. These three contexts are called ANY_COEFF, ANY_CHR_COEFF and ANY_CHR_AC_COEFF and are listed along with the other contexts in Table 4.2. The three contexts all have only two pre-defined symbols installed, which represent “YES“ and “NO.” When a video sequence is encoded macroblock by macroblock, before any transform coefficients are encoded, the coder checks if there are any non-zero transform coefficients in the macroblock. If this is not the case, the ANY_COEFF context is used to encode and transmit a “NO” to the decoder, and none of the transform coefficients are sent. When the decoder receives the “NO,” all the transform coefficients are automatically set to zero. On the other hand, if there exist non-zero transform coefficients in the macroblock, a “YES” is encoded and transmitted, and then follows the transform coefficients. The ANY_CHR_COEFF and ANY_CHR_AC_COEFF contexts are used in an

Table 4.2 Contexts for Model 2.

Context name	Context size	NOIS	Context description
MB_MODE_CONTEXT	8	99	Used to code MB_Type only
MVD_CONTEXT	60	373	Codes all the motion vector data
LUMA_CONTEXT_INTER	50	15165	Handles luma transf. coeff.
DC_CONTEXT_INTER	40	544	Handles chroma DC transf. coeff.
AC_CONTEXT_INTER	30	8161	Handles chroma AC transf. coeff.
ANY_COEFF	2	66	Codes status for all transf. coeff.
ANY_CHR_COEFF	2	39	Codes status for chroma transf. coeff.
ANY_CHR_AC_COEFF	2	11	Codes status for chr. AC transf. coeff.

analogous way. If there exist non-zero transform coefficients, the ANY_CHR_COEFF context is used to tell the decoder whether there are any non-zero chroma coefficients. If this is the case, the ANY_CHR_AC_COEFF context is then finally used to tell the decoder whether there are any non-zero chroma AC coefficients in the macroblock.

It is easy to see the similarity between the CBP codewords and the "YES" and "NO" symbols in the three ANY_COEFF, ANY_CHR_COEFF and ANY_CHR_AC_COEFF contexts. One difference, however, is that the three contexts work on a macroblock level instead of a 4x4 sub-block level like the CBP data type.

4.3.2 Initial Symbol Distribution for Model 2

The initial symbol distribution is identical to the one in Model 1 (Section 4.2.3), except of course for the three new contexts for which statistics were collected from the same frame and video sequence as for the other contexts. The number of symbols corresponding to the initial symbol distribution is presented in the third column in Table 4.2.

4.3.3 ADAPT_RATE for Model 2

The handling of transform coefficients in Model 2 results in considerably fewer symbols being sent from encoder to decoder compared to Model 1. Hence, also considerably less statistics are available for adaptation. To compensate for this the ADAPT_RATE was increased to 20. One drawback however, compared to Model 1, is that the number of symbols sent varies a great deal with quantization step and type of video sequence. An adaptation rate of 20 might not be optimal for all quantization steps and video sequences. Nevertheless, experiments showed that a lower adaptation rate almost always resulted in less efficient coding, and that an adaptation rate greater than 20 in most cases did not affect the coding efficiency much.

4.3.4 Bit-Rate Reduction for Model 2

The results for Model 2 are presented in Figure 4.6, 4.7 and 4.8. In Figure 4.6 and 4.7, showing the bit-rate reduction for the Foreman and the Mobile video sequence respectively, we see that Model 2 well outperforms Model 1. For the Mobile video sequence there is an average bit-rate reduction of approximately 5% for low bit-rates, and for high bit-rates the bit-rate reduction, although less than for Model 1, still exceeds 11%. However, the results for the Foreman video sequence are not remarkably positive, and according to the video codec evaluation in Section 2, it seems possible to design a model giving greater bit-rate reductions than Model 2 (compare Figure 2.13b). Figure 4.8, showing the bit-rate reduction for the Silent video sequence, further illustrates the weaknesses of Model 2.

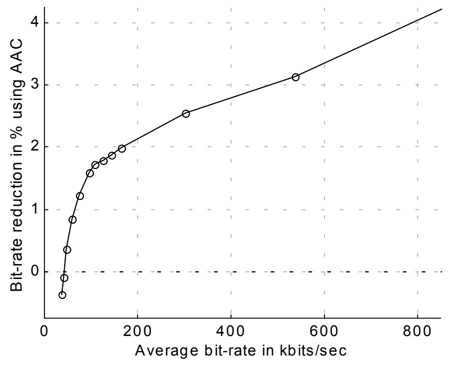

Figure 4.6 Bit-rate reduction for Foreman (QCIF)

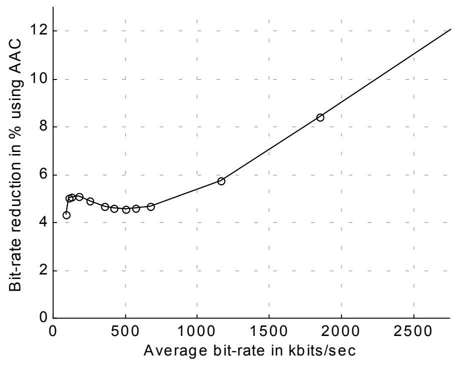

Figure 4.7 Bit-rate reduction for Mobile (QCIF)

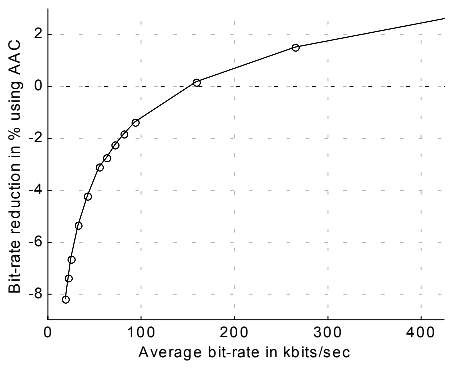

Figure 4.8 Bit-rate reduction for Silent (QCIF)

4.3.5 Conclusions

The main difference between Model 1 and Model 2 is the control of whether transform coefficients are transmitted. Sending fewer transform coefficients, as in Model 2, clearly makes the coding more efficient for low bit-rates compared to Model 1.

In the three new contexts ANY_COEFF, ANY_CHR_COEFF and ANY_CHR_AC_COEFF in Model 2, there is an inherently defined model for transform coefficient statistics. This model asserts for example that if one of the chroma transform coefficients is non-zero, it is more likely that also one of the luma transform coefficients is non-zero than vice versa. Models like this, built up from different contexts, and capturing spatially or temporally local statistics, are as explained in Section 4.1 one of the key ideas in adaptive arithmetic coding in combination with video coding and image coding. A major concern, however, is that models can be designed in infinitely many different ways. The most straightforward way to find a good model is to examine the statistics of the output of the video coder. Unfortunately these statistics are usually dependent on quantization step and type of video sequence.

One way to possibly improve the coding efficiency for the model inherently defined by the ANY_COEFF, ANY_CHR_COEFF and ANY_CHR_AC_COEFF contexts, is to tailor a rate-distortion function (see Section 1.3.1) to the model. Let us assume for instance, that no rate-distortion optimization takes place in the H.26L core and that we are faced with the extreme case that a particular macroblock contains only one non-zero transform coefficient, which happens to be a chroma AC coefficient. This means, for Model 2, that all the transform coefficients in the macroblock will be transmitted, compared to, if the single non-zero chroma AC coefficient had been zero, none of the transform coefficients would be transmitted. The question is: Is it reasonable to use as many bits to send all transform coefficients as to send only the non-zero chroma AC coefficient? This is the kind of questions answered by a typical rate-distortion method. It is difficult, however, to say how much a rate-distortion method could improve Model 2.

Worth mentioning is also that for both Model 1 and Model 2, experiments using end of block (EOB) symbols were conducted. That is, for each sub-block in each macroblock, an EOB symbol was sent if all the remaining transform coefficients in the sub-block were zero, and then the encoding continued with the next sub-block. However, neither for Model 1, nor for Model 2 did the use of EOB symbols give positive results when used for INTER-coded frames. This is in line with general adaptive arithmetic coding for which skewed symbol distributions give more efficient coding. The use of EOB symbols results in considerably fewer zero transform coefficients transmitted (zero transform coefficients are by far the most common), which in turn results in a more uniform symbol distribution giving less efficient coding.

The tests and evaluations so far have shown that for low bit-rates it seems preferable to send few symbols from the encoder to the decoder. As less transform coefficients are transmitted for Model 2 compared to Model 1, it becomes more important to use an appropriate typical symbol distribution when contexts are purged and re-initialized. One main difference between the H.26L model and Model 2, is that for the H.26L model, work has been done to establish a typical symbol distribution from a training set of video sequences. This typical symbol distribution is defined through the universal VLC table (see Section 1.2.6). The initial symbol distributions for the contexts in Model 2 are for simplicity established from a very small amount of statistics. Therefore, we could draw the conclusion that a poor initial symbol distribution is the reason for the negative results of Model 2. However, it is very difficult to say how true such a conclusion is. Establishing an appropriate initial symbol distribution for a model is a time consuming and cumbersome work, and if the model turns out to be inefficient the work was done in vain, since each model with different contexts has its unique symbol statistics.

If we design a third model built directly on the data types, or syntax elements in H.26L, we can adopt the symbol distribution defined through the universal VLC table as initial symbol distribution for the contexts. Intuitively, the results of such a model should be, in the worst case no bit-rate reduction, but in the average case some or much bit-rate reduction due to arithmetic coding and adaptation.

4.4 Model 3

Since the typical symbol distribution defined by the universal VLC table is based on the use of run-length coding, the latter needs to be put (back) into the modeling stage. All the data types are still, however, coded separately as shown in Figure 4.9.

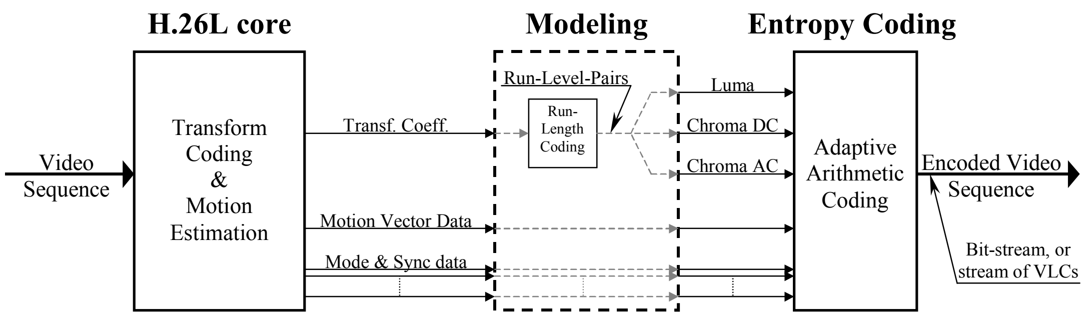

Figure 4.9 Schematic figure of the modeling process for Model 3.

4.4.1 Model 3 Contexts

As mentioned earlier, the contexts in Model 3 correspond exactly to the data types, or syntax elements used in the current version of H.26L. This allows us to use the already established typical symbol distribution defined through the universal VLC table, as initial symbol distribution for the contexts. Model 3 has the same contexts as Model 1, plus the CBP_CONTEXT_INTER context for the CBP data type. The contexts are listed in Table 4.3 (refer to Section 4.4.3 for an explanation of the “adapt. rate” column). The CBP_CONTEXT_INTER context, having size 48, includes all the possible symbols in the CBP

Table 4.3 Contexts for Model 3.

Context name	**Context size**	**NOIS**	**Adapt. rate**	**Context description**
MB_MODE_CONTEXT	8	113	10	Used to code MB_Type only
CBP_CONTEXT_INTER	48	2001	40	Used to code CBP data only
MVD_CONTEXT	100	8101	163	Codes all the motion vector data
LUMA_CONTEXT_INTER	300	130605	2621	Handles luma transf. coeff.
DC_CONTEXT_INTER	100	8101	163	Handles chroma DC transf. coeff.
AC_CONTEXT_INTER	100	8101	163	Handles chroma AC cransf. coeff.

data type. The transform coefficient contexts' sizes are increased to handle the new and different symbols due to the use of run-length coding. Run-length coding calls for larger contexts because, as mention in Section 4.2.1, a small number of unique symbols will be translated into a larger number of unique symbols.

The size of the MVD_CONTEXT has also been increased even though no modification is made to the coding of motion vectors. A larger MVD_CONTEXT increases the probability that costly escape symbols are avoided for lower bit-rates, for which a larger amount of motion vectors are used (see Figure 2.6).

4.4.2 Initial Symbol Distribution for Model 3

The initial symbol distribution used for all contexts is defined through the universal VLC table (see Section 1.2.6). The length c_s of a codeword corresponding to symbol s, gives the probability p_s for that symbol according to

$$p_s = 2^{-c_s} .$$

The probability of a codeword/symbol decreases with increasing codeword number. The size of a context defines how many of the codewords/symbols are installed in the context, starting with symbol 1, corresponding to codeword 0. The highest codeword number in a context is equal to the size of the context minus one. The smallest possible frequency count, i.e. 1, in the initial symbol distribution will correspond to the last codeword in the context, which has the smallest probability. All the other probabilities need to be normalized using a scaling constant called SCALE_CONST, which transforms probabilities into frequency counts. The SCALE_CONST for a context with N symbols, i.e. a context of size N, is calculated as

$$\text{SCALE\_CONST} = 2^{C_{N-1}} ,$$

where c_x is the length of codeword x.

For instance, if the context size $N = 8$, then $c_{N-1} = c_{8-1} = c_7 = 7$ (according to Figure 1.4b). Thus, the SCALE_CONST = 2^7 = 128. Table 4.4 shows codewords, probabilities for the corresponding symbols and the frequency count in the initial symbol distribution for each codeword/symbol. The frequency count is the probability multiplied by SCALE_CONST. Since the size of the MB_MODE_CONTEXT is 8, Table 4.4 shows exactly the codewords and the initial symbol distribution for the MB_MODE_CONTEXT.

A consequence of the procedure for how the initial symbol distribution is established is that the number of symbols corresponding to the initial symbol distribution (NOIS), becomes a function of the context size N. NOIS is the sum of the frequency counts, which is approximately equal to SCALE_CONST.

Table 4.4 Codewords, probabilities and frequency counts for a context of size 8.

Codeword no.	Codeword	Probability	Freq. count
0	1	0.5	64
1	0 0 1	0.125	16
2	0 1 1	0.125	16
3	0 0 0 0 1	0.03125	4
4	0 0 0 1 1	0.03125	4
5	0 1 0 0 1	0.03125	4
6	0 1 0 1 1	0.03125	4
7	0 0 0 0 0 0 1	0.0078125	1

4.4.3 Improved Weighting of Symbols

The fact that the NOIS differs much between contexts and that for some contexts the NOIS is great, calls for a small modification of the adaptive arithmetic codec. The ADAPT_RATE variable has been replaced with three variables called ADAPT_RATE_MIN, ADAPT_RATE_MAX and INV_ADAPT_RATE. In Model 3, the adaptation rate is different for each context (see Table 4.3) and it is calculated as

$$\text{Adaptation Rate} = \frac{\text{SCALE\_CONST}}{\text{INV\_ADAPT\_RATE}}.$$

This essentially means that the INV_ADAPT_RATE will correspond to the number of symbols that needs to be encoded before the symbol distribution built up from adaptation is weighted as heavy as the initial symbol distribution. Two thresholds, ADAPT_RATE_MIN and ADAPT_RATE_MAX gives the user possibility to set the minimum and maximum adaptation rate independent of the INV_ADAPT_RATE.

Experiments were made using different values for the three parameters. Figure 4.10 gives some guidance. Here is presented the average number of symbols per frame for three different video sequences, which increases almost linearly with increasing bit-rate (decreasing quantization step). The fact that the average number of symbols per frame varies a great deal with bit-rate makes it more difficult to choose an appropriate adaptation rate (see discussion about future work in Section 5.2.2). In Model 3, INV_ADAPT_RATE, ADAPT_RATE_MIN and ADAPT_RATE_MAX were set to 50, 10 and 3000 respectively. This gives a quick adaptation, which is preferable for very low bit-rates. Another example of how the three parameters could be set is 50, 20 and 400 respectively, which gives a slight increase in coding efficiency for low and high bit-rates, but performs somewhat worse for very low bit-rates.

The adaptation rates for the contexts in Model 3 are presented in Table 4.3. Only the MB_MODE_CONTEXT's adaptation rate is affected by one of the threshold values.

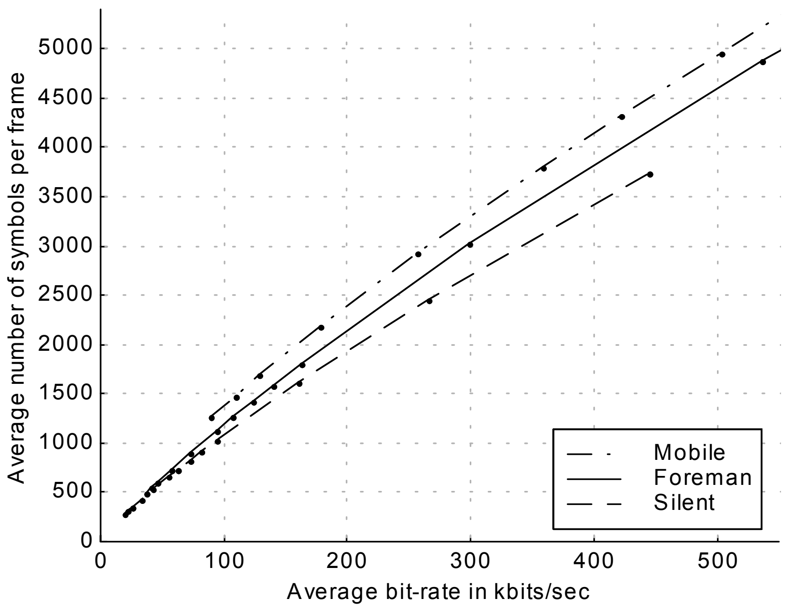

Figure 4.10 Average number of symbols per frame for the Mobile, Foreman and Silent video sequences.

4.4.4 Modified Handling of Psync and Ptype Codeword and Escape Symbols

Not mentioned so far is that for Model 1 and 2, the adaptive arithmetic codec was stopped and restarted once every frame and once for each escape symbol sent. This was done so that the Psync and Ptype codewords could be transmitted using the H.26L universal VLC table, and so that escape symbols could be sent using normal binary representation. (Note: Sending escape symbols and the standard H.26L Psync and Ptype codewords *could* be done without stopping and restarting the adaptive arithmetic codec, if some major modifications were made to the arithmetic encoding and decoding core algorithms. These modifications would not be trivial, however, and would not allow the separation between the H.26L core, the modeling and the adaptive arithmetic codec.)

Since the cost of stopping and restarting the adaptive arithmetic codec is 32 bits (see Section 3.2.5), the above handling of Psync and Ptype codewords and escape symbols is probably inefficient for low- or very low bit-rates and for cases where a large amount of escape symbols is used.

In Model 3 the adaptive arithmetic codec is never stopped and restarted. Instead the information in the Psync and Ptype codewords is sent using appropriate contexts (these contexts' symbols correspond more or less to the data and will not be discussed further). Escape symbols are "spelled out" using a context containing the 10 digits in the decimal system (i.e. 0,1,2,...,9). Synchronization without stopping and restarting the adaptive arithmetic codec can be implemented if the synchronization process is integrated with the arithmetic codec. However, this is not implemented in the current version of the adaptive arithmetic codec and instead approximately 8 bits per frame is added to the resulting bit-rates to compensate for the lack of picture synchronization codewords.

4.4.5 Bit-Rate Reduction for Model 3

Figure 4.11a, 4.12a and 4.13a show the bit-rate reduction for Model 3 for the Foreman, Mobile and Silent video sequences in the QCIF format, respectively. Included are also the results for the CIF format in Figure 4.11b, 4.12b and 4.13b.

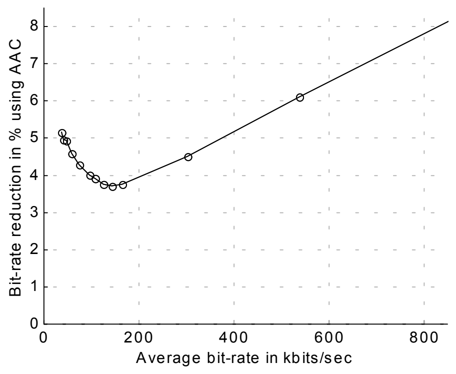

Figure 4.11a Bit-rate reduction for Foreman (QCIF)

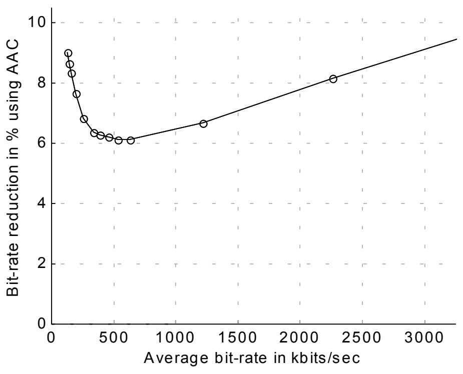

Figure 4.11b Bit-rate reduction for Foreman (CIF)

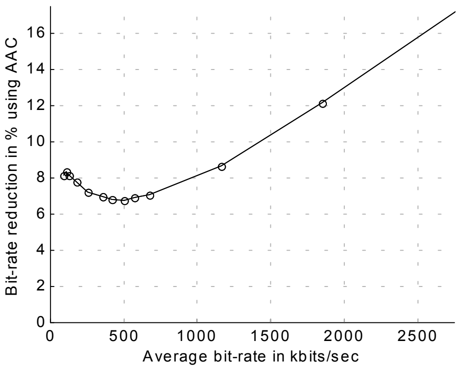

Figure 4.12a Bit-rate reduction for Mobile (QCIF)

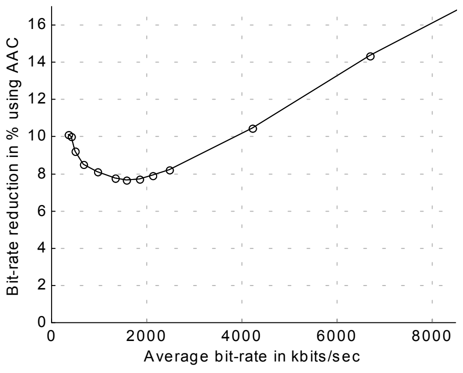

Figure 4.12b Bit-rate reduction for Mobile (CIF)

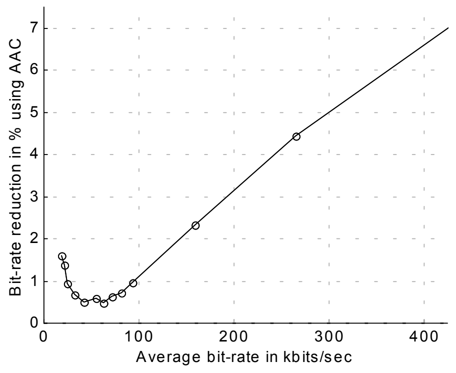

Figure 4.13a Bit-rate reduction for Silent (QCIF)

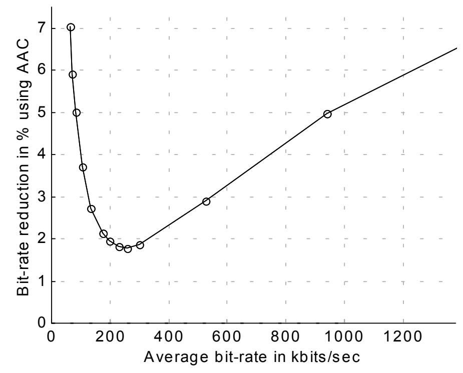

Figure 4.13b Bit-rate reduction for Silent (CIF)

As seen in the figures above, Model 3 gives better results than both Model 1 and 2. The slope of the curves is also kept for higher bit-rates for all video sequences. For instance, the reduction in bit-rate for the Mobile video sequence, in both the QCIF and CIF format is as high as 20% for very high bit-rates (quantization steps around 3).

Important to notice in Figures 4.11-4.13 is also how much the efficiency of Model 3 varies with quantization step and type of video sequence.

4.4.6 Conclusions

From the results of Model 1 and 2, we drew the conclusion that it is, regardless of the nature of the model, important to have a good initial symbol distribution for each context, and that it is especially true for lower bit-rates. In order to fully exploit the typical symbol distribution already defined by the universal VLC table, Model 3 was designed using contexts corresponding exactly to the original H.26L data types. The results of Model 3 show a significant improvement over Model 1 and 2, especially for lower bit-rates. Thus, the hypothesis that having a good initial symbol distribution is of great importance, certainly appears to be true.

There are several reasons why the curves in Figure 4.11-4.13 look like they do. The main reason why the bit-rate reduction differs with type of video sequence is because video sequences containing less motion, such as the Silent video sequence, produce much fewer symbols for the adaptive arithmetic codec to adapt to, and thus yield less efficient coding. The bit-rate reductions also vary with quantization step a great deal. This is due to two things. First of all, a lower quantization step means more symbols, and thus more statistics to adapt to. Second, the H.26L universal VLC table is optimized for a quantization step of about 19 (compare Figure 2.13). Hence, it is easier to improve the coding for bit-rates corresponding to quantization steps below and above 19, for which the original H.26L codec is not optimized. There is also another important reason, however, for the improved coding for very low bit-rates, which is due to the modified handling of synchronization codewords (Psync). The amount of bits used for synchronization is not dependent on quantization step, and thus coding synchronization codewords with fewer bits only affects the compression efficiency notably for very low bit-rates.

According to Figure 4.11-4.13, the bit-rate reduction is greater for video sequences in the CIF format compared to the same video sequences in the QCIF format. This is in line with the theory that more symbols give better adaptation and thus a more efficient adaptive arithmetic coding. It is somewhat surprising, however, that the difference is not greater considering the amount of additional symbols sent from encoder to decoder for video sequences in the CIF format compared to the QCIF format.

5. Conclusions and Future Work

Combining adaptive arithmetic coding and video coding, in this case video coding in H.26L, has been shown to involve problems that are not trivial to solve. Many of the conclusions drawn from the work presented in this document can be found throughout the text, and more specifically in Section 4.2.6, 4.3.5 and 4.4.6 covering the conclusions for the three different models designed. However, in this section follows a general discussion about adaptive arithmetic coding in combination with the H.26L video codec.

The section ends with a discussion regarding future work. Here, rate-distortion optimization is discussed, and ideas concerning a fourth model are presented.

5.1 Conclusions

Due to the wide variety of applications for video coding, all having different requirements and demands, one can not give a simple “yes” or “no” answer to the question if adaptive arithmetic coding and video coding in H.26L is a good combination. Instead, we need to look at the benefits and drawbacks separately, and then later, when we have knowledge about the exact requirements and demands, make a decision whether or not it would be beneficial to employ adaptive arithmetic coding.

5.1.1 Amount of Statistics Crucial for Good Adaptation

It is very clear, from every test made, that the more statistics available for adaptation, the more can be gained using adaptive arithmetic coding. The method described in this document can, without further improvements, be used to enhance the coding in H.26L considerably for high bit-rate environments. The method also shows improvements for lower bit-rates, albeit not as positive.

5.1.2 Complexity

A concern about adaptive arithmetic coding in general is the higher complexity compared to other entropy coding techniques such as VLCs or arithmetic coding without adaptation. We also need extra storage in both encoder and decoder to store the contexts’ initial symbol distributions. In the current version of the adaptive arithmetic codec, however, the initial symbol distributions are generated during encoding and decoding, and therefore do not require any extra space for storage. This is not a desirable solution, however, since it most certainly slows down the encoding and decoding a great deal.

An important benefit of the presented methods and models is that no additional information needs to be transmitted from the encoder to the decoder.

5.1.3 Error Resilience

Possible difficulties to combine error resilience and adaptive arithmetic coding have been mentioned several times throughout the report. There are no actual reasons, however, why error resilience methods like forward error correction, data partitioning, header extension codes, etc. (see Section 1.3.2) would be more difficult to *implement* in a video codec using adaptive arithmetic coding compared to a video codec using VLCs. The main concern, however, lies in how much error resilience methods interfere with the adaptation process, and thus how much of the efficiency won through adaptation would be lost if an error resilience method was employed. We have learnt that the efficiency of the adaptation is more or less a function of the amount of symbols, or statistics, in between purging and re-initialization of contexts. Therefore, the negative effects in coding efficiency when employing error resilience methods are entirely dependent on the scheme's necessity for purging and re-initialization, and the frequency of the latter.

Once again we need to know the requirements and demands on the particular system before we can say that adaptive arithmetic coding is bad in an error resilience point of view. What can be said, however, is that an error prone environment in general has great negative effects on the adaptation process.

5.1.4 A Few Words About Speed

Not mentioned so far is the performance in speed of the adaptive arithmetic codec. This is because the work presented in this document concerns the efficiency of the H.26L entropy coding solely, and issues like speed has not been taken into account. Nevertheless, here are a few properties regarding speed for the original adaptive arithmetic codec presented in [22], which the adaptive arithmetic codec is based on.

The tree-based structure used to maintain cumulative frequencies for the symbols in a context takes O(log N) time per symbol to access, where N is the size of the context. This allows for a small look-up cost even when the context size N is large. The tree-based structure is also space-economical, and requires only N words of storage.

The current version of the adaptive arithmetic codec uses, as mentioned in Section 3.2.2, 30 bits to store the total number of frequency counts in a context. Using as much as 30 bits for the frequency counts allows faster execution, since much of the arithmetic can be done to a low precision. Furthermore, the adaptive arithmetic codec uses shift/add integer arithmetic in its calculations instead of multiplications and divisions (compare [10]). The low precision that is required allows, on some architectures, faster execution with shift/add operations.

5.2 Future Work

The large amount of possibilities for model design and the nature of the parameters that can be set for the adaptive arithmetic codec certainly allow much work and research in the future.

We could for example construct a video codec that employs different entropy coding techniques for different bit-rates. The video codec could for high (and possibly intermediate) bit-rates employ adaptive arithmetic coding giving considerable bit-rate savings, and some other entropy coding method for lower bit-rates.

Another example is to go deeper into the H.26L core (see Figure 2.8) and adapt to statistics from smaller elements than transform coefficients and motion vectors. Such a technique could become even more efficient if combined with a sophisticated motion model (see Section 1.3.3), where the performance of the adaptive arithmetic coding technique is optimized for that particular model.

5.2.1 Rate-Distortion Optimization

Rate-distortion optimization takes place in the H.26L core. One of the rate-distortion methods in H.26L concerns the choice of macroblock mode for INTER-pictures, i.e. the number of motion vectors per macroblock or INTRA-coded macroblock (see Section 2.3.3). The number of bits (the rate) used for different macroblock modes is known to the encoder. When a picture is being encoded, different macroblock predictions are acquired from a previous picture using different macroblock modes. The quality (the distortion) of each prediction is calculated using the SAD metric (see Section 1.1.3). Having knowledge about the number of bits for each macroblock mode and the SAD values for different predictions, the encoder chooses a macroblock mode according to the desired rate-distortion ratio (compare Section 1.3.1 and Figure 1.7). A rate-distortion method is, according to Section 1.3.1, supposed to calculate the distortion (in this case the quality of a prediction) from the **reconstructed** picture. The above simplification, to use the original picture instead, might decrease accuracy of the rate-distortion method but allows faster execution.

The rate-distortion optimization described above is based on H.26L's universal VLC table, and does not necessarily optimize the rate-distortion ratio for H.26L with adaptive arithmetic coding. However, the rate-distortion method is still employed, but for simplicity the possibility to use INTRA-coded macroblocks in INTER-pictures has been disabled as mentioned in Section 4.2.2.

To optimize the above rate-distortion method for H.26L with adaptive arithmetic coding, we need to take into account that the number of bits (the rate) required to code a symbol is dependent on the state of the symbol distribution in the symbol's context. The symbol distribution is in turn dependent on the initial symbol distribution and which symbols have been sent since the last purge and re-initialization of the context. The nature of the adaptive arithmetic codec tells us that common symbols are less costly to code. Hence, the above rate-distortion method for H.26L with adaptive arithmetic coding should, in a list, rank the symbols in the context according to frequency, and then use the list to calculate the rate, which is used in the rate-distortion method to choose macroblock mode.

It is difficult to say how much a rate-distortion method based on adaptive arithmetic coding would improve the results presented in this document. In fact, not using INTRA-coded macroblocks in INTER-pictures might degrade the efficiency of the original H.26L codec as much as a rate-distortion method optimized for adaptive arithmetic coding would improve the efficiency for a H.26L codec that employs adaptive arithmetic coding. In that case, the possible improvements are already included in the results in this document. However, experiments show that, for instance the Foreman video sequence uses about 2% INTRA-coded macroblocks in INTER-frames for low and intermediate bit-rates and about 1% for high bit-rates. With this in mind, it seems like the degradations in the original H.26L codec due to unemployment of INTRA-coded macroblocks in INTER-pictures should be small. On the other hand, different types of applications/environments may use/demand more than 2% INTRA-coded macroblocks in INTER-pictures.

5.2.2 Model 4

Using the conclusions from Model 1, 2 and 3, ideas for a fourth model have come up. Model 4 should be designed according to the following considerations:

1. Use appropriate but few contexts. These contexts should be designed using techniques discussed in Section 4.1 to exploit spatial redundancies within images better. For example: the blocks in Figure 5.1 below could represent macroblocks, or alternatively smaller sub-blocks. When the transform coefficients in block *x* are encoded, one out of three different contexts is used. The context used depends on the transform coefficients

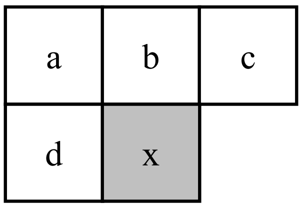

Figure 5.1 Block *x* and adjacent blocks *a*, *b*, *c* and *d*.

in block *a*, *b*, *c* and *d*. We could for example look at the size (level) of transform coefficients or the amount of non-zero transform coefficients in block *a*, *b*, *c* and *d*, and use this as a basis for the choice of context. It is important, however, to keep down the number of contexts since each context needs its share of statistics, and statistic resources are easily exhausted, especially for low bit-rates.

2. For each quantization step, or for a few appropriate groups of quantization steps, do the following (this could also possibly be done for different video formats):

- Construct an initial symbol distribution for each context, using an appropriate training set of video sequences.
- Find optimal context sizes (for example, the distribution of transform coefficients is dependent on quantization step, and smaller contexts are preferable for higher quantization steps and vice versa).
- Purge and re-initialize contexts more frequently for lower quantization steps and less frequently for higher quantization steps for which less statistics is available for adaptation.
- Find optimal adaptation rate (with help from for example statistics similar to the statistics in Figure 4.10).

Model 4 would improve the results from Model 3, however, it is difficult to say by how much. If the Model 4 contexts are created for use inside the H.26L core and not designed for the output symbols of the H.26L core (compare Figure 2.8), the additional compression improvements for Model 4 compared to Model 3 would most likely be even greater.

Appendix A – Acronyms and Abbreviations

AAC	Adaptive Arithmetic Coding
ACL	Average Codeword Length
ADE	Average Disjoint Entropy
CBP	Coded Block Pattern
CIF	Common Intermediate Format
DCT	Discrete Cosine Transform
DSR	Dynamic Symbol Reordering
EOB	End Of Block
EOS	End Of Sequence
FEC	Forward Error Correction
HDTV	High Definition TV
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
JPEG	Joint Photographic Experts Group
MB	Macroblock
MPEG	Moving Picture Experts Group
MSE	Mean Square Error
MV	Motion Vector
MVD	Motion Vector Data
NOIS	Number Of Initial Symbols
QCIF	Quarter Common Intermediate Format
R-D	Rate-Distortion
RVLC	Reversible Variable Length Coding
SAD	Sum of Absolute Differences
SNR	Signal to Noise Ratio
TC	Transform Coefficient
VLC	Variable Length Coding

References

[1] Aalmoes, Roalt. 1996. "*Video compression techniques over low-bandwidth lines*." Netherlands: Twente University.

[2] Bailey, Donald G. and Body, Nick B. 1998. "*Efficient Representation and Decoding of Static Huffman Code Tables in a Very Low Bit Rate Environment*." Proceedings ICIP98. IEEE Comput. Soc. 3 vol.

[3] Bell, Timothy C., Cleary, John G. and Witten, Ian H. 1990. "*Text Compression*." New Jersey: Prentice-Hall, Inc.

[4] Bichsel, Martin and Ohnesorge, Krystyna W. 1994. "*Fast Adaptive Arithmetic Coding*." SPIE Vol. 2186 Image and Video Compression.

[5] Brusewitz, Harald. 1987. "*Arithmetic Coding - A Tutorial*." Report no. TRITA-TTT-8701. Stockholm.

[6] Chai, Bing-Bing, Palaniappan, Kannappan, Vass, Jozsef and Zhuang, Xinhua. 1999. "*Significance-Linked Connected Component Analysis for Very Low Bit-Rate Wavelet Video Coding*." IEEE Transactions on Circuits and Systems for Video Technology, Vol. 9, No. 4, June 1999.

[7] Chistopoulos, Charis (editor), "*JPEG2000 Verification Model 5.0*." ISO/IEC JTC/SC29/WG1 N1422, August 1999.

[8] Clearly, John G., Neal, Radford M. and Witten, Ian H. 1987. "*Arithmetic Coding for Data Compression*." Communications of the ACM, Vol. 30, No. 6, June 1987.

[9] Girod, Bernd and Hartung, Frank. 1999. "*Improved Encoding of DCT Coefficients for Low Bit-Rate Video Coding Using Multiple VLC Tables*." Telecommunications Laboratory, University of Erlangen-Nuremberg 1999.

[10] Huynh, Linh and Moffat, Alistair. 1997. "*A Probability-Ratio Approach to Approximate Binary Arithmetic Coding*." IEEE Transactions on Information Theory, Vol. 43, No. 5, September 1997.

[11] ISO/IEC 14496-2. "*Information Technology – Generic Coding of Audio Visual Objects, Part 2: Visual*." Final Draft of International Standard, 18 December 1998.

[12] ISO/IEC JTC1/SC29 WG11. "*The MPEG Home Page*." Online. Available: http://www.cselt.it/mpeg. 29 October 1999.

[13] ITU-T. 1995. "*Video Coding for Low Bitrate Communication, Draft ITU-T Recommendation H.263*." International Telecommunication Union.

[14] ITU-T. 1998. "*Response to Call for Proposals for H.26L*." International Telecommunication Union, Sixth Meeting: Seol, Korea, November 1998.

[15] ITU-T. 1999. "*Enhanced Samsung Entropy Coding*." International Telecommunication Union, Ninth Meeting: Red Bank, New Jersey, October 1999.

[16] ITU-T. 1999. "*H.26L Test Model Long Term Number 1 (TML-1) draft 2*." International Telecommunication Union.

[17] ITU-T 1999. "*Meeting Report of the Eighth Meeting (Meeting H) of the ITU-T Q.15/16 Advanced Video Coding Experts Group – Berlin, Germany, 03-06 August 1999*." International Telecommunication Union, August 1999.

[18] ITU-T 2000. "*MVC Decoder Description*." International Telecommunication Union, Geneva, February 2000.

[19] Konrad, Janusz and Stiller, Christoph. 1999. "*Estimating Motion in Image Sequences*." IEEE Signal Processing Magazine, July 1999.

[20] Langdon, Glen G., Jr. 1991. "*Adaptive Binary Arithmetic Coding for Multi-media Applications*." IEEE Comput. Soc. Press, Digest of Papers.

[21] Li, Weiping and Ling, Fan. 1998. "*Dimensional Adaptive Arithmetic Coding for Image Compression*." Proceedings of the 1998 IEEE International Symposium on Circuits and Systems.

[22] Moffat, Alistair, Neal, Radford M. and Witten, Ian H. 1996. "*Arithmetic Coding Revisited*." Preliminary presented at IEEE Data Compression Conference, Snowbird, Utah, March 1995.

[23] Netravali, Arun N. and Haskell, Barry G. 1995. "*Digital Pictures*." New York: AT&T Bell Laboratories, Plenum Press.

[24] Ortega, Antonio and Ramchandran, Kannan. 1998. "*Rate-Distortion Methods for Image and Video Compression*." IEEE Signal Processing Magazine, November 1998.

[25] Pennebaker, William B. 1998. "*Tracking Nonstationary Probabilities in Adaptive Binary Arithmetic Coding*." IEEE Transactions on Image Processing, Vol. 7, No. 12, December 1998.

[26] Rabbini, Majid and Jones, Paul W. 1991. "*Digital Image Compression Techniques*." Washington: The Society of Photo-Optical Instrumentation Engineers.

[27] Strintzis, M. G. and Triantafyllidis, G. A. 1999. "*A Context Based Adaptive Arithmetic Coding Technique for Lossless Image Compression*." IEEE Signal Processing Letters, Vol. 6, No. 7, July 1999.

[28] Talluri, Raj. 1998. "*Error-Resilient Video Coding in the ISO MPEG-4 Standard*." IEEE Communications Magazine, June 1998.

[29] Watkinson, John. 1995. "*Compression in Audio & Video*." ISBN 0-240-51394-0. Oxford: Butterworth-Heinemann Ltd.